Generative Story Worlds as Linear Logic Programs

Chris Martens
Carnegie Mellon University
cmartens@cs.cmu.edu

João F. Ferreira
Teesside University
J.Ferreira@tees.ac.uk

Anne-Gwenn Bosser
ENI Brest
Lab-STICC UMR6285
bosser@enib.fr

Marc Cavazza
Teesside University
M.O.Cavazza@tees.ac.uk

Abstract

Linear logic programming languages have been identified in prior work as viable for specifying stories and analyzing their causal structure. We investigate the use of such a language for specifying story *worlds*, or settings where generalized narrative actions have uniform effects (not specific to a particular set of characters or setting elements), which may create emergent behavior through feedback loops.

We show a sizable example of a story world specified in the language Celf and discuss its interpretation as a story-generating program, a simulation, and an interactive narrative. Further, we show that the causal analysis tools available by virtue of using a proof-theoretic language for specification can assist the author in reasoning about the structure and consequences of emergent stories.

Introduction

Linear logic (Girard 1987) (LL) has been proposed as a suitable conceptual framework to specify and reason about interactive narratives (Bosser, Cavazza, and Champagnat 2010), which has led to a variety of applications such as narrative analysis (Bosser et al. 2011), narrative generation (Martens et al. 2013), and authoring and validation tools (Dang et al. 2011). At the same time, the success of logically-motivated languages such as Inform 7[1] together with the documented interest of Interactive Fiction (IF) writers for emergent narratives[2] suggests that the IF community may welcome a programming language approach to the authoring of rule-based, generative story worlds.

Our work proposes the use of a logic programming language for specifying emergent systems representing narrative worlds. We use a language based on constructive LL, which supports action description, use of resources, and changes imposed on the story world, and offers a basis for *analysis* of storylines and causal relationships, all within the same theory of proof. This approach lets us have our cake (in the sense of designing narratives with richly-interacting processes) and eat it, too (in the sense of predicting and controlling the consequences of such processes).

Related Work

Planning systems have been widely adopted for the construction of interactive narratives (Young 1999; Porteous, Cavazza, and Charles 2010), mostly because of their support for the representation of causality. It is generally accepted that LL is a strong candidate for such a representation (Girard and Lafont 1987), and a proof in LL can be equated to a plan (Masseron 1993; Masseron, Tollu, and Vauzeilles 1993). Aside from surface differences, such as LL predicates having meaningful multiplicity, the main benefit of using LL is that planning actions, plan synthesis, and plans themselves can all be accounted for uniformly under a minimal theory of inference and proof.

Standard logic programming approaches to narratives provide evidence for the concise and readable nature of declarative specification (Grasbon and Braun 2001; Lang 1999) but lack the native ability to model state and causality afforded by LL. One logic programming engine for managing generativity in games uses the Event Calculus (EC) to overcome this shortcoming (Smith and Mateas 2011).

LL has previously been explored for game analysis, as a front-end for Petri nets (Collé, Champagnat, and Prigent 2005). Further development demonstrated the value in using a *constructive* formulation of LL to directly correlate the formal notion of *story* with the notion of logical proof (Bosser, Cavazza, and Champagnat 2010; Bosser et al. 2011). Dang investigated LL for complete exploration and validation of scenarios for educational purposes (Dang, Champagnat, and Augeraud 2013). However, by relying on a backward-chaining proof search interpretation, these works have favored authorial intent above narrative emergence. Our contribution is to investigate LL for modeling story scenarios which are primarily exploratory and generative, rather than purely goal-driven, using forward-chaining proof search. This duality between intent (backward chaining) and exploration (forward chaining) has previously been used for combining deliberative and reactive behaviour for agent modelling (Harland and Winikoff 2004). Our language of investigation, called Celf (Schack-Nielsen and Schürmann 2008), supports both forward and backward chaining through the

[1]http://www.inform7.com

[2]See for instance Emily Short's article on emergence in narrative interaction design at http://emshort.wordpress.com/2013/02/14/introducing-versu/

focusing theory of proof search (Chaudhuri, Pfenning, and Price 2008).

Contributions

Expanding on our previous work (Martens et al. 2013), we demonstrate how to encode more general story settings that are *parametric* over arbitrary world states, settings, and characters. We demonstrate that these rules create *nontrivial feedback loops* in terms of their causal structure, allowing for long and perhaps unpredictable chains of events to create pivotal events in stories.

We further extend our use of tools to this case, including those we get for free from the language Celf as well as the extrinsic graphical tool and query language. These tools illustrate the use of proof term structure in exploring the consequences of rules which may have been conceived in the haphazard, ad-hoc way of first draft designs.

The takeaway point of this paper is that **encoding story worlds in a linear logic programming language allows for informative exploration of their consequences.** For the remainder of the paper, we will support this claim by describing an example and its semantics, demonstrating how nondeterministic proof search can be used to *generate* stories, and sketching our work in progress on tools for *analyzing* and *interacting with* story models.

System Overview

The basis of our framework is the use of LL as a language for specifying actions. This setup is similar to the use of planning languages such as STRIPS (Fikes and Nilsson 1971) in the sense that we model the world as a collection of predicates and specify the available actions declaratively in terms of those predicates.

A *world state* is an unordered collection of predicates Δ, and actions are written as state transitions $A \multimap \{B\}$, signifying an action that replaces the state components described by A with those described by B.

The process of execution (or *simulation* or *generation*) of a story, then, is to take a user-specified initial state Δ_0, find some rule r in the specification Σ that can apply, and apply it to get Δ_1. This process is iterated until no further rules apply, at which point we say we have reached *quiescence*.

When a rule $r : A \multimap \{B\}$ is applied on state Δ, A, the next state is Δ, B. The component Δ which is irrelevant to the rule stays the same, solving the *frame problem* present in other settings, such as event and situation calculi (Hayes 1971). This property arises from the inference rules defining $\multimap$ as the proposition-level internalization of logical entailment.

We summarize the connectives and notation used below:

abstract syntax	concrete syntax	meaning
$A \multimap \{B\}$	`A -o {B}`	replacement
$A \otimes B$	`A * B`	conjunction of resources
$!A$	`!A`	persistent resource

Persistent resources may be regarded as permanent facts as in standard propositional logics.

Example: A Romantic Tragedy Story World

The key to programming with LL is formulating one's problem in terms of *resources*, i.e. the components of a story that may interact and change. Then, the author describes *how* (through what actions) those components change. In a bit more detail, the process for designing a story world takes the following shape:

1. Identify the components of story state, such as physical location and character relationships. Declare a predicate (type) for each of these; for example, declare that `anger C C′` is a well-formed state component when C and C' are characters. We map predicates in our example to their intended meanings in the table below.
2. Identify the *narrative actions*, i.e. ways that characters can interact with each other or their environs to cause changes in the state. For instance, a rule for flirting with a character other than one's lover may cause anger in the lover directed at both flirters, which is represented by the rule

```
do/flirt/conflict
: eros Flirter Flirtee * eros Other Flirtee
  -o {eros Flirter Flirtee * eros Flirtee Flirter
      anger Other Flirter * anger Other Flirtee}.
```

   (The words beginning with capital letters, by convention, mark logic variables, and are implicitly quantified at the beginning of the rule—the names themselves are arbitrary.)

A complete story world specification is simply a collection of these predicate and rule declarations. The author may then join it with its complementary half, a specification of setting elements, characters, and initial states, to generate stories or analyze the system we have described. The remainder of this section carries out an example, a case study of a Shakespeare-inspired romantic tragedy world, following the aforementioned workflow.[3]

The state we model includes sentiments between characters (several forms of love and hate), physical locations and basic movement, possession of key objects (such as weapons), relationship states (marriage and being single), desires for objects, and solitary emotions (depression).

Predicate	Meaning
`at C L`	C is (alive) in location L
`has C O`	C possesses an object O
`neutral C C′`	C feels neutrally toward C'
`philia C C′`	C feels affection toward C'
`anger C C′`	C feels anger toward C'
`eros C C′`	C feels attraction toward C'
`unmarried C`	C is unmarried
`married C C′`	C is married to C'
`depressed C`	C is depressed
`suicidal C`	C is suicidal
`!dead C`	C is dead
`!murdered C C′`	C murdered C'
`!actor C`	C is a character in the story

[3]The complete, runnable code for this example can be found at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/tragedy.clf`.

These predicates do not specify a particular cast of characters or setting, but they could be said to pertain to a particular *genre* of story, in this case romance and tragedy. The delineation of the story world into predicates is an act of careful human design, often iterated with the generation of stories, and the choices made here make all the difference in terms of which actions are possible to write and therefore what shapes the story can take. For instance, the choice to include two kinds of love, `eros` and `philia`, means that we can codify social rules about sexual relationships between characters. Similarly, the choice *not* to include a predicate for a character's gender renders it impossible to enforce heteronormative relationships. We have to make a choice about whether we want to model a realistic description of human behavior (which often contradicts social norms) or enforce social norms by constraining actions with our formal description.

A selection of rules for our chosen genre is given below, starting with the rules for basic social interaction:[4]

```
do/formOpinion/like
: at C L * at C' L *
  neutral C C'
  -o {at C L * at C' L * philia C C'}.

do/formOpinion/dislike
: at C L * at C' L *
  neutral C C'
  -o {at C L * at C' L * anger C C'}.

do/compliment/private
: at C L * at C' L * philia C C'
  -o {at C L * at C' L * philia C C' * philia C' C}.

do/compliment/witnessed
: at C L * at C' L * at Witness L * philia C C' *
  anger Witness C'
  -o {at C L * at C' L * at Witness L * philia C C' *
      anger Witness C' * philia C' C * anger Witness C}.

do/insult/private
: at C L * at C' L * anger C C'
  -o {at C L * at C' L * anger C C' * anger C' C *
      depressed C'}.

do/insult/witnessed
: at C L * at C' L * at Witness L *
  anger C C' * philia Witness C'
  -o {at C L * at C' L * at Witness L *
      anger C C' * philia Witness C' * anger C' C *
      depressed C' * anger Witness C}.

mixed_feelings
: at C L * anger C C' * philia C C' -o {at C L * neutral C C'}.
```

Note that there are two "versions" each of the actions for insulting and complimenting, one that happens "in private" and another that affects a witness in the same location. This encoding signals a weakness: it is unnatural to represent a *broadcast* of an action affecting every character who might be in range, since LL primarily codifies *local* state changes. On the other hand, this mechanism's nondeterminism could be argued to reflect the chance involved in whether an action goes noticed.

Next we codify the rules for romantic interaction, which include tranformations between eros and philia as well as flirting, marriage, and divorce:

```
do/fallInLove
: at C L * at C' L' *
  eros C C'
  -o {at C L * at C' L' * eros C C' * philia C C'}.

do/eroticize
: at C L * at C' L' *
  philia C C' * philia C C' * philia C C' * philia C C'
  -o {at C L * at C' L' * philia C C' * eros C C'}.

do/flirt/ok
: at C L * at C' L * eros C C' * unmarried C * unmarried C'
               -o {eros C C' * eros C' C *
                    unmarried C * unmarried C' *
                    at C L * at C' L}.
do/flirt/discreet
: at C L * at C' L * eros C C'
  -o {eros C C' * eros C' C * at C L * at C' L}.

do/flirt/conflict
: at C L * at C' L * at C'' L *
  eros C C' * eros C'' C
                -o {eros C C' * eros C' C * eros C'' C
                    * anger C'' C' * anger C'' C
                    * at C L * at C' L * at C'' L}.
do/marry
: at C L * at C' L *
  eros C C' * philia C C' *
  eros C' C * philia C' C *
  unmarried C * unmarried C'
  -o {married C C' * married C' C * at C L * at C' L *
      eros C C' * eros C' C * philia C C' * philia C' C }.

do/divorce
: at C L * at C' L' *
  married C C' * married C' C * anger C C' * anger C C'
  -o {anger C C' * anger C' C * unmarried C * unmarried C'
      * at C L * at C' L'}.

do/widow
: married C C' * at C L * dead C'
  -o {unmarried C * at C L}.
```

These rules include the generation of sentiments from a netural stance, transformations between the two kinds of love `philia` and `eros`, and flirting, which strengthens both kinds of love, but causes anger if witnessed by another paramour. We also include rules that modify the marriages of characters.

Next we supply rules governing death and violence:

```
do/murder
: anger C C' * anger C C' * anger C C' * anger C C' *
  at C L * at C' L  * has C weapon
  -o {at C L * !dead C' * !murdered C C' * has C weapon}.

do/becomeSuicidal
: at C L *
  depressed C * depressed C * depressed C * depressed C
  -o {at C L * suicidal C * wants C weapon}.

do/comfort
: at C L * at C' L *
  suicidal C' * philia C C' * philia C' C
  -o {at C L * at C' L *
      philia C C' * philia C' C * philia C' C}.

do/suicide
: at C L * suicidal C * has C weapon -o {!dead C}.

do/mourn
: at C L * philia C C' * dead C'
  -o {at C L * depressed C * depressed C}.

do/thinkVengefully
: at C L * at Killer L' *
  philia C Dead * murdered Killer Dead
  -o {at C L * at Killer L' * philia C Dead *
      anger C Killer * anger C Killer}.
```

The violence module introduces several potential feedback loops between murder and vengeance, suicide, mourning, and depression.

Finally, we have a few actions that can affect possession:

```
do/give
: at C L * at C' L * has C O * wants C' O * philia C C'

do/steal
: at C L * at C' L * has C O * wants C' O
  -o {at C L * at C' L * has C' O * anger C C'}.
```

[4]The location predicates `at C L` in these rules signify not just location, but also that the character in question is alive and available in the story. The `at` atom is consumed when a character dies.

```
do/loot
: at C L * dead C' * has C' O * wants C O
  -o {at C L * has C O}.
```

Given this set of rules, we note that the story world is multi-agent in nature—the interactor with such a story doesn't obviously "play" one particular character, and the rules aren't defined as "behaviors" attached to a given agent. They portray the interiority of all characters at once, allowing them to be referenced and changed in combination.

Initial State

After describing the general rules of our Shakespearean tragedy story world, which are parameterized over characters and locations, we can fill in specific elements, such as the characters and setting of Romeo and Juliet, to have a complete and runnable specification.

First we can describe the *persistent* (unchanging, i.e. not linear) facts about the story, in this case the world map and the cast of characters:

```
mon/town : accessible mon_house town.
town/mon : accessible town mon_house.
cap/town : accessible cap_house town.
town/cap : accessible town cap_house.

a-romeo : actor romeo.
a-juliet : actor juliet.
a-montague : actor montague.
a-capulet : actor capulet.
a-mercutio : actor mercutio.
a-nurse : actor nurse.
a-tybalt : actor tybalt.
a-apothecary : actor apothecary.
```

Next, we need to designate the *initial* state of all the linear predicates. It could look something like this:

```
story_start :
init -o { at romeo town * at montague mon_house * at capulet cap_house
  * at mercutio town * at nurse cap_house * at juliet town
  * at tybalt town * at apothecary town

  * has tybalt weapon * has romeo weapon * has apothecary weapon

  * unmarried romeo * unmarried juliet
  * unmarried nurse * unmarried mercutio * unmarried tybalt
  * unmarried apothecary

  * anger montague capulet * anger capulet montague
  * anger tybalt romeo * anger capulet romeo * anger montague tybalt

  * philia mercutio romeo * philia romeo mercutio
  * philia montague romeo * philia capulet juliet
  * philia juliet nurse * philia nurse juliet

  * neutral nurse romeo
  * neutral mercutio juliet * neutral juliet mercutio
  * neutral apothecary nurse * neutral nurse apothecary}.
```

The first two groups of atoms describe the story world locations where the characters begin and their possessions. The next group represents which characters are unmarried at the start of the story. The next two groups represent existing relationships (sentiments) among characters, and finally the last group represents which characters haven't met each other yet (and so feel neutrally towards each other).

Serendipity

While general-purpose, generative story rules serve to create an emergent notion of story, one might argue that in many stories, *coincidence* or *serendipity* plays a large role in the story being interesting. We have a way to codify the idea of serendipitous events as well: it is that *rules* of the form $A \multimap \{B\}$ can be treated on the same level as other propositions, and thus included in initial states.

For example, we encode Romeo and Juliet's "love at first sight" as a single rule added to the consequent of `story_start`:

```
story_start :
init -o {
  ...
  {Forall L. at romeo L * at juliet L
      -o {eros romeo juliet * at romeo L * at juliet L}}
  ...}.
```

This single-use rule dictates that whenever Romeo and Juliet are in the same place, Romeo may gain eros for Juliet. The construct `Forall L ...` allows the rule to be parameterized over the shared location where they meet. (This use of quantification differs from the implicit quantification happening on the outside of every rule; here, we want to bind the location *locally* rather than over the entire `story_start` rule.)

Final States

We could at this point call the example complete and ask the system to begin executing the rules from the given initial state. But before doing so, we might be interested in some high-level structural questions, such as: do stories in this specification *end*? If so, how do they end?

Answering this question requires thinking about the rules *operationally* rather than as static descriptions of possible actions. The operational semantics of the program is based on the formal description given in the section System Overview wherein a *step* is an evolution of a context Δ, A to a context Δ, B along a rule $A \multimap \{B\}$.[5] The simulation terminates when no more steps can be made, i.e. when no more rules apply to the current context.

We make the following observation: all of the character actions in our specification require at least one character to be alive (represented by the `at` predicate). Most of the rules preserve location/aliveness of the characters, but the actions corresponding to character deaths (murder and suicide) do not. Therefore, the story will terminate when all characters have died.

At several points in the story, multiple rules will apply. If we consider a *fair* operational semantics, i.e. one where when multiple rules apply, each has some positive chance of being chosen, then *eventually* the termination condition will be reached. How long the story goes on before it ends is a function of the more specific probabilities a rule is selected with in the story engine – which property is not defined by the language semantics, but is observable of the implementation.

For this example, we decided to make termination conditions that would be met more frequently for the sake of shorter stories, more dense with interesting behavior. This also allows us to codify a set of "story endings." To do this, we create new atoms `final` and `nonfinal`, add `nonfinal` to the initial state, and write a few rules that

[5]This is a simplification of the story—the true transition semantics are given by rules that do not require the rule's precondition to be already formed in the context, but may perform backward search to find it.

lead from desired end conditions to the `final` atom, consuming `nonfinal`, e.g.:

```
ending_happy
: nonfinal *
  actor C * actor C' *
  at C L * at C' L * married C C'
  -o {final}.

ending_vengeance
: nonfinal *
  actor C1 * actor C2 * actor C3 *
  killed C1 C2 * philia C3 C2 * killed C3 C1
    -o {final}.
```

Proofs as Stories

To reiterate, the takeaway point of this paper is that *encoding story worlds in a linear logic programming language allows for informative exploration of their consequences*. Two of these exploration techniques are *random story generation* and *structural analysis*, and what enables those techniques to fall naturally out of our encoding is the fact that there is a direct and formal correspondence between *stories* and *proofs*.

Once we write our specification as a collection of logical formulas, we can initiate a *query* such as

```
?- init -o {final}.
```

This query asks whether there is a *proof* from the state described by `init` to the atom `final`. Initially, proof search is *goal-directed* or backward-chaining: it adds `init` to the current state and considers how to prove `{final}`. Before now, we have been treating the curly braces `{-}` as meaningless, but they mean something in terms of proof search: it must now switch to a *forward-chaining* or *generative* mode, running inference forward from the `init` atom. Only once the entire story has terminated will it look for `final`, at which point search will succeed if it finds it.

But the point of executing the query isn't really to find out whether the initial state leads to a valid conclusion. What we are interested in is the *trace* generated by execution—the witness to the validity of the proposition, i.e. the proof!

Interpreting propositions as types, we assign a proof term to the implication $A \multimap B$ as a λ-term, or function, from terms of type A to terms of type B. Within the body M of the function $\lambda x{:}A.M$, the story term can make use of the variable x, e.g. by projecting out its components and applying rules in the story signature to them.

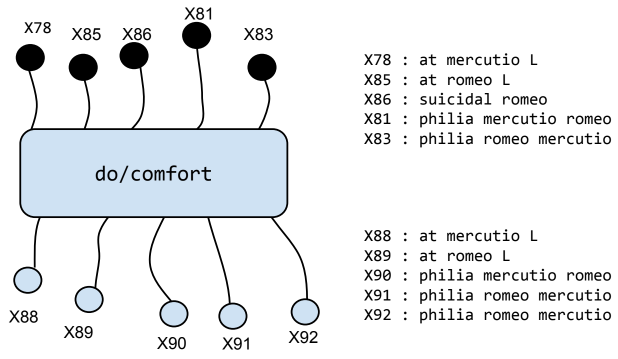
Proofs of monadic goals, such as, in our example, `{final}`, are lists of let-bindings that capture the trace of actions, e.g.:

```
...
let {[X73, [X74, [X75, [X76, X77]]]]}
  = do/insult/private [a-tybalt, [a-romeo, [X68, [X66, X72]]]] in
let {[X85, [X86, X87]]}
  = do/becomeSuicidal [a-romeo, [X79, [X41, [X59, [X52, X77]]]]] in
let {[X88, [X89, [X90, [X91, X92]]]]}
  = do/comfort [a-mercutio, [a-romeo,
      [X78, [X85, [X86, [X81, X83]]]]]] in
let {[X101, [!X102, [!X103, X104]]]}
  = do/murder
    [a-romeo, [a-tybalt,
      [X58, [X40, [X76, [X51, [X94, [X96, X27]]]]]]]] in
let {[X105, [X106, [X107, X108]]]}
  = do/compliment/private
    [a-nurse, [a-juliet, [X46, [X47, X30]]]]] in
let {[X109, [X110, [X111, X112]]]}
  = do/compliment/private
    [a-juliet, [a-nurse, [X106, [X105, X108]]]]] in
let {[X113, X114]}
  = do/loot [a-romeo, [a-tybalt, [X101, [X102, [X26, X87]]]]] in
...
```

This fragment of trace shows how certain scenes, such as one wherein Tybalt drives Romeo to murder with Mercutio's support, are interleaved with independent scenes, such as a loving conversation between the Nurse and Juliet.

Within the let-binding portion of the trace, we see a record of the story rules selected, which can be seen as a linear progression of events. But additionally, each binding `let {[X1, ..., Xn]} = rule [Y1, ..., Ym]` represents the call of `rule` on previously-generated resources `Y1 ... Ym` representing its antecedent, generating *new* resources `X1 ... Xn` representing its consequent. This allows us to analyze the proof term for dependency structure, specifically revealing which events are *in*depedenent and can be thought of as concurrent storylines.

Here is a graphical depiction of the let-binding for Mercutio comforting Romeo in the above trace fragment:


X78
X85
X86
X81
X83
do/comfort
X88
X89
X90
X91
X92
X78 : at mercutio L
X85 : at romeo L
X86 : suicidal romeo
X81 : philia mercutio romeo
X83 : philia romeo mercutio
X88 : at mercutio L
X89 : at romeo L
X90 : philia mercutio romeo
X91 : philia romeo mercutio
X92 : philia romeo mercutio

The variables at the top of the image represent the inputs/antecedents to the rule, including the persistent witnesses of Romeo and Mercutio's actorhood, which arise from the signature, as well as those generated by prior let-bindings. The variables in the bottom are freshly generated by the rule, representing its outputs/consequents.

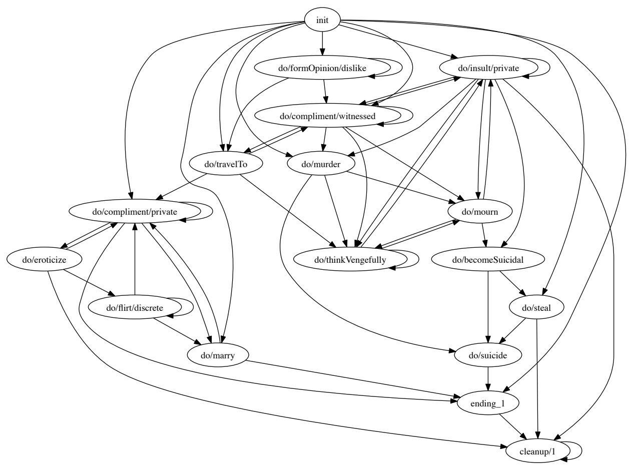
Although the let-bindings appear presented linearly, because they encode data dependency information, independent forks can actually be extricated from one another through a notion of *concurrent equality* derived from the syntactic structure, i.e. `let x1 = M1 in let x2 = M2 in M` may be considered the same trace as `let x2 = M2 in let x1 = M2 in M` iff the inputs of `M2` are separate from the outputs of `M1`.

It is important to emphasize the critical use of Celf's basis in constructive logic, which enables the identification of program execution with a structured witness to examine, complete with dependency structure.

Graphical Analysis

For the sake of making the dependency structure more visually explicit, we have been developing a tool that automatically translates generated stories into causal diagrams in the form of directed graphs. Nodes of the graph are actions in the story, and edges can be viewed as causal relationships between story events involving those actions.

This tool takes a proof term and the story specification as inputs, and it generates the extrapolation of the let-binding illustration given in the previous section to the whole trace: we wind up with a causal network of action nodes representing the story's nonlinear progression of events, e.g.:



The tool also has an interface for *queries* on sets of traces. E.g. the query `exists ending_1` would tell us the set of stories with `ending_1` (a marriage and a death). This tool allows us to test our specifications for how closely they match our authorial intent. For instance, if we want to find out if any stories contain unfulfilled vengeance, we might issue the query

```
exists do/thinkVengefully && ~link do/thinkVengefully do/murder
```

(read as: there exists a "think vengefully" action, but there is no link between "think vengefully" and "murder") which might tell us that no stories satisfy the predicate. If we intend for vengeance to occur more or less frequently, we may tune the parameters of the rules (e.g. how many `anger` atoms it generates) and run the query again.

Ongoing Work

Suppose we wanted to treat the same logic program not as a random story generator but as an *interactive* story. At a first cut, a player could interact with proof search by selecting a rule whenever proof search would otherwise select one randomly between several that apply. But in the interest of creating a more exploratory feel, rather than present a finite set of available story branches, we wish to follow the interactive fiction tradition of allowing a general *grammar* of actions. The action language for our example might correspond loosely to the available rules, e.g.

```
insult(C, C')  compliment(C, C')  flirt(C, C')  marry(C, C')
murder(C, C')  comfort(C, C')  suicide(C)  divorce(C, C')
```

Parameterizing actions over all of the characters they concern supposes an omnicient point of view where the interactor may control all characters. If we suppose a single-character point of view, then the first parameter `C` in the actions above becomes implicit. After giving this language of actions, we can modify the original program so that the rules concerning, e.g., flirtation, take an extra precondition that an atom `do(flirt(C, C'))` appears. This atom should be linear so that the rule only fires once.

We are working on an extension to the core language underlying Celf that allows the introduction of action atoms by the player once the program has reached quiescence, which we believe soundly captures the idea of interpreter and interactor taking turns. We formalize this idea of turn-taking as a language construct called *phases* (Martens 2013), which turn out to be generally useful for other engineering concerns, such as structuring a program into independent (module-like) components. Further, phases enable us to describe the *broadcast* behavior we were missing earlier: broadcast can be its own phase that iterates a rule to quiescence over all occupants of a room. After the introduction of phases, we believe the language will be suitable as the core of a system for authoring large, expressive works of interactive fiction and exploratory interactive simulation ("sandbox" games). In the context of a larger project, of course, we would also need tools for parsing and generating natural language.

We are also developing a more complete suite of analysis capabilities, akin to model checking, that would allow authors to predict and control the range of narrative possibilities, perhaps choosing to restrict them to a more linear causality tree (in the case of a rigid narrative) or relax the constraints (in the case of a sandbox world). We are designing a verification-like system overlaying the language wherein the author may, on a per-phase basis, write down properties (invariants, preconditions, and postconditions) to the logic program, which may then be automatically checked. An application of this functionality might include ensuring as an invariant that, whenever there is a locked door, a key is reachable from the player's location.

Summary of Contributions

We see this work as making the following contributions to narrative technologies:

- A programming language-based approach to authoring interactive stories, founded on a theory which yields an interpretation of stories as logical proofs.
- Demonstration of how said interpretation enables the *generation*, *analysis*, and *interactive interpretation* of stories from story worlds.
- Atop prior work in formalizing stories in linear logic, we show that this approach can be used not only for *validating* goal-driven story interactions but also for sculpting exploratory, reactive experiences.

Ongoing work should allow for combining exploratory and verificational approaches to authorship, laying the theoretical groundwork for a language for generative interactive fiction.

Acklowledgments

The authors thank the anonymous reviewers for their thorough and encouraging feedback. The first author thanks Frank Pfenning, Martin van Velsen, and Adam Smith for conversations that helped contextualize our work.

References

Bosser, A.-G.; Courtieu, P.; Forest, J.; and Cavazza, M. 2011. Structural analysis of narratives with the Coq proof assistant. In *ITP*.

Bosser, A.-G.; Cavazza, M.; and Champagnat, R. 2010. Linear Logic for non-linear storytelling. In *ECAI 2010*, volume 215 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Chaudhuri, K.; Pfenning, F.; and Price, G. 2008. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning* 40(2–3):133–177.

Collé, F.; Champagnat, R.; and Prigent, A. 2005. Scenario analysis based on linear logic. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE '05. New York, NY, USA: ACM.

Dang, K. D.; Hoffmann, S.; Champagnat, R.; and Spierling, U. 2011. How authors benefit from linear logic in the authoring process of interactive storyworlds. In *ICIDS*, 249–260.

Dang, K. D.; Champagnat, R.; and Augeraud, M. 2013. A methodology to validate interactive storytelling scenarios in linear logic. *T. Edutainment* 10:53–82.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, 608–620. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Girard, J.-Y., and Lafont, Y. 1987. Linear Logic and lazy computation. In Ehrig, H.; Kowalski, R.; Levi, G.; and Montanari, U., eds., *TAPSOFT '87*, volume 250 of *LNCS*. Springer Berlin / Heidelberg. 52–66.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science* 50(1):1–102.

Grasbon, D., and Braun, N. 2001. A morphological approach to interactive storytelling. In *Proceedings of the Conference on Artistic, Cultural and Scientific Aspects of Experimental Media Spaces (cast01)*.

Harland, J., and Winikoff, M. 2004. Agents via mixed-mode computation in linear logic. *Annals of Mathematics and Artificial Intelligence* 42:167–196.

Hayes, P. J. 1971. *The Frame Problem and Related Problems on Artificial Intelligence*. Stanford University.

Lang, R. R. 1999. A declarative model for simple narratives. In *Narrative Intelligence: Papers from the AAAI Fall Symposium*. AAAI Press.

Martens, C.; Bosser, A.-G.; Ferreira, J. F.; and Cavazza, M. 2013. Linear logic programming for narrative generation. In *Logic Programming and Nonmonotonic Reasoning 2013*.

Martens, C. 2013. Logical interactive programming for narrative worlds. PhD thesis proposal, Carnegie Mellon University.

Masseron, M.; Tollu, C.; and Vauzeilles, J. 1993. Generating plans in Linear Logic: I. Actions as proofs. *Theoretical Computer Science* 113(2):349–370.

Masseron, M. 1993. Generating plans in Linear Logic: II. A geometry of conjunctive actions. *Theoretical Computer Science* 113(2):371–375.

Porteous, J.; Cavazza, M.; and Charles, F. 2010. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Trans. Intell. Syst. Technol.* 1(2):10:1–10:21.

Schack-Nielsen, A., and Schürmann, C. 2008. Celf — a logical framework for deductive and concurrent systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'08)*, 320–326. Springer LNCS 5195.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Trans. Comput. Intellig. and AI in Games* 3(3):187–200.

Young, R. M. 1999. Notes on the use of plan structures in the creation of interactive plot. In *Narrative Intelligence: Papers from the AAAI Fall Symposium*. AAAI Press.