Linear Logic Programming for Narrative Generation

Anne-Gwenn Bosser, João F. Ferreira, and Marc Cavazza
Teesside University

Chris Martens
Carnegie Mellon University

Abstract. In this paper, we explore the use of Linear Logic programming for story generation. We use the language Celf to represent narrative knowledge, and its own querying mechanism to generate story instances, through a number of proof terms. Each proof term obtained is used, through a resource-flow analysis, to build a directed graph where nodes are narrative actions and edges represent inferred causality relationships. Such graphs represent narrative plots structured by narrative causality. Building on previous work evidencing the suitability of Linear Logic as a conceptual model of action and change for narratives, we explore the conditions under which these representations can be operationalized through Linear Logic Programming techniques. This approach is a candidate technique for narrative generation which unifies declarative representations and generation via query and deduction mechanisms.

Keywords: Linear Logic Programming, Narrative Modelling, Celf

1 Introduction

Linear Logic [1] has recently been proposed as a suitable representational model for narratives [2]: its resource-sensitive nature [3] allows to naturally reason about narrative actions and the changes they cause in the environment.

In this paper, we explore Linear Logic programming as a tool for narrative representation and narrative generation.We describe how initial circumstances and narrative actions can be declared in the Linear Logic programming language Celf [4] and how using Celf's search mechanism allows the generation of proof terms which can be interpreted as causally structured narrative plots. To improve narrative analysis, we developed a prototype front-end to Celf which allows the generation and analysis of such plots.

Preliminary results are encouraging, allowing the generation of story variants through a methodical programming approach.

After presenting related works and introducing the language Celf and our dedicated plot interpretation and analysis tool, we describe how to use story material to program and generate a variety of plots using the novel *Madame Bovary* [5]: its narrative causal structure has been emphasized in Flaubert's working material [6].

2 Related Works

Computational Approaches to Narrative Modelling. Narratives have always been an important topic for research in AI for their role as knowledge structures [7]. Typical models to reason about these structures include Situation Calculus [8, 9], Event Calculus [10], and other non-standard models [11].

Recent years have seen the widespread adoption of a variety of planning techniques for the construction of narrative generation systems [12, 13], mostly because of their support for the representation of causality. Linear Logic provides an expressive model of action and change (see [14]), where information revision is dealt with at the level of the logical rules through the use of linear implication. It is generally accepted that Linear Logic ,particularly Intuitionistic Linear Logic [3],is a strong candidate to represent causality. A proof in Linear Logic sequent calculus can be equated to a plan [15],which has led previous work to explore its suitability for narrative representation, using a story-as-proof analogy [2]. This support of narrative causality at the logical level (which does not require any adaptation of the language as in [16]) is also the advantage of *Linear* Logic programming approaches when compared with standard logic programming approaches to narrative generation such as [17] or [18].

Related Applications of Linear Logic. In [19], a fragment of Linear Logic is used to model game scenarii and a transformation into Petri nets is then used for game properties validation. A similar modelling and validation approach has used a wider fragment to assist the design of piloting systems for interactive stories [20], which have similar but more generic properties than computer games. A semi-automated application of Linear Logic to narratives has been described in [21]: the intractability of proof search in the most expressive fragments of Linear Logic has led the authors to use the Coq proof assistant for story generation and for evidencing properties transcending all narratives.

Linear Logic Programming. While classical logic provides systems for constructing and deconstructing *truth*, *linear* logic is a deductive system for constructing and deconstructing *resources* making its computational interpretations particularly well suited to tackle resource-sensitive problems. A few programming systems are based on Linear Logic. Lolli [22] follows goal-directed backward proof-search interpretation in the intuitionistic fragment of Linear Logic (differing in this among other traits from Lygon [23] which allows multiple goals). LolliMon [24] and Celf [4] are more recent systems that have extended Lolli and where the forward and backward chaining phases may be controlled by the programmer using a monad.

Linear Logic programming has been used in domains such as natural language processing and language specification (see [25] for an overview and application survey). To the best of our knowledge, the present work constitutes the first application of Linear Logic programming to narrative generation.

3 Celf Programming

Celf [4] is a Linear Logic programming system in the LF family [26]. At its core is an inference engine based on that of LolliMon [24], where forward and backward chaining phases may be controlled by the programmer. Atop that structure, Celf uses dependent types for the representation of logical predicates; this approach to logic programming means that the result of a query is a *term* of the corresponding *type*, which can be analysed as a computational artefact. We provide here an overview of Celf[1] and show how Celf programs are structured, focusing on the aspects relevant for narrative generation.

Celf programs are normally divided into two main parts: **a signature**, which is a declaration of type and terms constants describing data and transitions, and **query directives**, defining the problem for which Celf will try to find solutions (proof terms showing that a given type is inhabited).

The technique used by Celf to compute proof terms is called focusing, based on the foundations of *Focused Linear Logic* [27] interpreted as Monadic Concurrent Logic Programming. [24]

3.1 Focused Linear Logic

Focusing is a proof search strategy which was introduced by Andreoli [27]. It reduces the search space to structural normal forms of proofs without reducing the expressiveness of their computational interpretations.

Focusing requires that we split the connectives (and thus formulae depending on their main connective) of Linear Logic into two groups, *synchronous* and *asynchronous*. Asynchronous formulae, when they are in the goal position of a sequent, may be eagerly decomposed without backtracking. Decomposing a synchronous formula, on the other hand, requires *making a choice*, e.g. about how to split the context. A search algorithm may then have to backtrack in order to find the choice allowing the proof search to complete.

Focusing imposes a discipline on proof search that consists of phases: first applying all possible invertible rules, then making a choice of formulae in the sequent (asynchronous formulae on the left or synchronous formulae on the right) to *focus* on. Focusing on a formula means decomposing it and focusing on its subformulae until this process reaches an atomic proposition, uninterrupted by rules that may apply to formulae out of focus.

3.2 Forward Inference via Monadic Logic Programming

The fragment of Linear Logic restricted only to asynchronous connectives yields a backward-chaining, Prolog-like semantics (as implemented in Lolli [22]). On the other hand, synchronous connectives enable us to write rules that function as *transitions* and yield a forward-chaining, Datalog-like semantics. Inspired by

[1] The Celf system can be obtained from https://github.com/clf/celf

LolliMon [24], Celf gives the programmer control over when to enter a forward-chaining phase, which may use synchronous connectives, through the use of a monad. In Celf, monadic encapsulation is denoted using curly brackets $\{\ldots\}$.

For example, any rule of the form $A \multimap \{S\}$ may fire when and only when we are in a forward chaining phase, whereas rules of the form $A \multimap B$, where B is *not* a monadic expression, may take place only in the backward chaining phase. The phases can interact when the subgoals (or antecedents) of rules in one phase invoke predicates involving the other phase.

The search triggered by a query in Celf begins in a backward-chaining phase using the query type as its goal, and if that type includes a monadic expression, it will enter a forward-chaining phase. This phase is implemented with a committed choice semantics, backtracking over the selection of a rule only when its antecedents cannot be met—effectively inducing a random choice between all fireable rules on each forward chaining step. This built-in nondeterminism is what lets us go automatically from a *specification* of a narrative structure to the automatic generation of stories.

3.3 A Celf Program

Figure 1 shows an example of a Celf program, representative of the form we use to model narratives. The program signature is structured as follows:

Atomic types (lines 2–10) correspond to atomic resources in the narrative.

Asynchronous types (lines 13–16) consist here of linear implications. Their general form is expressed in Celf as `A1 * ... * An -o {B1 * ... * Bn}`, where `*` denotes multiplicative conjunction and `-o` linear implication. Resources on the left are consumed and resources on the right produced. Note that, while when on the left-side of a linear implication the `*` can be treated without the monad (by interpreting `(A1 * ... * An) -o B` as syntactic sugar for the curried equivalent `A1 -o ... -o An -o B`), on the right of the `-o`, we need to encapsulate the conjunction in a monad, as explained previously in Section 3.2.

The narrative actions described here make use of *affine* resources of the form `@A`: such resources, introduced in the environment, *can* be consumed during the proof search at most once. By default, resources are *linear*, which means that they *need* to be consumed for any proof search to terminate.

Linear implications describe transitions and correspond to narrative actions. For example, line 14 of Figure 1,

```
emmaReadsNovel : type = emma * novel -o {@emma * @escapism}.
```

models a narrative action called `emmaReadsNovel` and expresses that upon consumption of the resource `novel`, the resource `escapism` is produced. It also expresses explicitly the conservation of the resource `emma`.

The initial state (lines 20–23) is a multiplicative conjunction (thus encapsulated in a monad as well) representing the initial state of the narrative

```
1  % Narrative resources
2  convent : type.
3  education : type.
4  grace : type.
5  novel : type.
6  ball : type.
7  escapism : type.
8  emma : type.
9  charles : type.
10 emmaCharlesMarried : type.
11
12 % Narrative actions
13 emmaSpendsYearsInConvent : type = emma * convent -o {@emma * !grace * !education
       }.
14 emmaReadsNovel : type = emma * novel -o {@emma * @escapism}.
15 emmaGoesToBall : type = emma * ball -o {@emma * @escapism}.
16 emmaMarries : type = emma * escapism * charles * grace -o {@emma * @charles
17                                                         * @emmaCharlesMarried}.
18
19 % Initial environment (resources + actions)
20 init : type = { @emma * convent * !novel * @charles * @ball
21               * @emmaSpendsYearsInConvent * emmaReadsNovel
22               * @emmaMarries * @emmaGoesToBall
23               }.
24
25 % Celf query
26 #query * * * 3 (init -o {emmaCharlesMarried}).
```

Fig. 1. Example of a Celf program: definition of four narrative actions in terms of narrative resources, of an initial environment, and of a query which will search 3 times for a solution to produce the resource `emmaCharlesMarried`. Comments start with `%`.

and available narrative actions. These can be defined as a) linear, like `emmaReadsNovel`: there is one copy in the initial environment, and it will need to be consumed for any computation to terminate successfully; b) affine, like `@charles`: there is initially one copy in the initial environment and it may or may not be consumed by a successful computation; c) persistent, like `!novel`: there is a generator for arbitrarily many copies in the initial environment and any number of them may be consumed by a successful computation.

The second part of the program consists of a query directive (line 26), defining the problem for which Celf will try to find solutions. Celf queries have five parameters. In this example, we just define the last two. The fifth argument, `init -o {emmaCharlesMarried}`, can be described as the goal. In other words, Celf will attempt to find a proof term showing that it is possible to produce the resource `emmaCharlesMarried` from the initial state defined as `init`. The fourth argument specifies the number of times the search must be repeated (here 3).

4 Well-formed Plots: Exhibiting Narrative Causal Structures

Following a long tradition of analysing causality via graphs [28], we developed a prototype tool, *CelfToGraph*[2], that automatically transforms proof terms generated by Celf into directed acyclic graphs. Such graphs represent narrative plots, structured by narrative causality, where nodes are narrative actions and edges represent inferred causality relationships. In this section, we present the structure of these graphs by using an example (Figure 2) generated from the program shown in Figure 1.

Looking at the program, we start by observing that the goal, `emmaCharlesMarried`, can only be produced by the action `emmaMarries`. Moreover, this action requires the existence of the resource `grace`, which can only be produced by the action `emmaSpendsYearsInConvent`. As a result, all the graphs will have the action `emmaSpendsYearsInConvent` as a predecessor of the action `emmaMarries`. On the other hand, the resource `escapism` can be produced by two different actions: `emmaGoesToBall` and `emmaReadsNovel`. This means that in some of the generated stories, the action `emmaMarries` can choose between the resource produced by these two actions. We use OR nodes to express choice of resources, as shown in Figure 2. As an aid to the reader, we label the edges with the names of the resources enabling the causality relation. (This is different from the default behaviour of *CelfToGraph*, where edges are not labelled.) Every structured graph has a node labelled `init` representing the initial narrative environment.

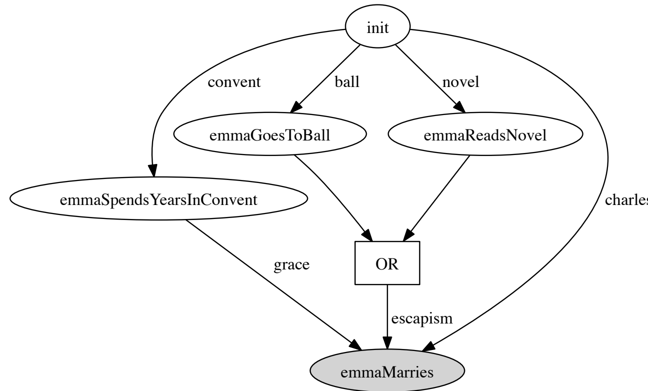

Fig. 2. Narrative plot generated from the Celf program presented in Figure 1. The node `init` represents the initial narrative environment. All the other nodes are narrative actions and edges represent inferred causality relationships.

[2] *CelfToGraph* requires Celf v2.9 and is available at https://github.com/jff/TeLLer

As shown in Figure 2, these graphs support the **characterisation of multiple causes**: if actions `emmaGoesToBall` and `emmaReadsNovel` happen before `emmaMarries`, it is impossible to determine which one of the first two is the real cause. The graphs generated are dependent on the actual unfolded sequence of actions, but it is important to note that there is not a bijection between graphs and Celf solutions: whilst a Celf solution is precise about which narrative resources are consumed, the graphs abstract that information and collect multiple choices in the form of OR nodes. Also, different orderings of the same narrative actions can produce the same graph. This means that the graphs generated allow the introduction of a very fine **differentiation between stories**: a different ordering of the same narrative actions during the unfolding of the story constitute a different story if and only if the inferred causal model of the narrative is different.

CelfToGraph also allows us to query the generated graphs. The two main query commands are `exists a`, which can be used to determine if the action `a` is in all the generated graphs, and `link a1 a2`, which can be used to determine if there is a link from action `a1` to action `a2` in all the graphs. These two commands can be composed using the usual boolean operators.

5 Programming a Narrative

Having introduced the canonical structure of a Celf program through some narrative examples, we can now describe the overall programming process through which an entire narrative is modelled and simulated using the Celf language built-in mechanisms.

5.1 Identification of Narrative Elements

The process of programming a narrative is that of describing circumstances that can, by execution of the program, generate one or many stories. Following a widespread paradigm in narrative generation research, we use a existing, linear, baseline story to support our experiments. The detailed formalization of such a story will bring to existence the many decision points and opportunities for actions to succeed, fail or be deferred. Identifying the circumstances within a static story such as Madame Bovary [5] is a human activity that can be assisted by companion works [6].

The narrative elements we need to identify and model fall into two main categories. **Narrative resources** are available story elements (including characters) as well as states of the story, which may be related to characters and motives. In the present example, we model them using atomic types. **Narrative actions** are transforming events occurring in the narrative. We model the impact they have on the narrative, in terms of narrative resource creation and consumption.

Consider the narrative action of Emma marrying Charles which will allow us to illustrate how we model resources. This action requires the presence of Emma and Charles, and some facts representing those characters' motivation for this action: Emma's grace and escapism. Their marriage results in the new fact that they are married and Emma gets bored. We write it in Celf as:

```
1  %% Encoding of an extract of the novel: Madame Bovary.
2  emma : type.
3  charles : type.
4  homais : type.
5  leon : type.
6  rodolphe : type.
7  emmaCharlesMarried : type.
8  convent : type.
9
10 <............>
11
12 emmaIsDespaired : type.
13 charlesIsConcerned : type.
14 emmaInLove : type.
15 leonEmmaTogether : type.
16 arsenic : type.
17 inheritance : type.
18 denounced : type.
19 ruin : type.
20 emmaIsDead : type.
21
22 emmaSpendsYearsInConvent : type = emma * convent -o {!novels * !grace * !education * @emma}.
23 emmaReadsRomanticNovels : type = emma * novels -o {@escapism * @escapism * @emma}. %remove one
       diminishes amount of stories
24 emmaMarriesCharles  : type = emma * escapism * grace * charles -o {emmaIsBored * @emma *!
       emmaCharlesMarried}.
25 emmaInvitedToBall : type = emma * emmaCharlesMarried * grace -o {@ball * @emma}.
26 emmaGoesToBall : type = emma * ball * escapism -o {@escapism * @escapism * @escapism * @escapism *
       @emma}.
27 emmaDoesNotGoToBall : type = emma * ball -o {emmaIsBored * @emma}.
28
29 < ......>
30
31 emmaJumpsThroughWindow : type = emma * emmaIsDespaired * emmaRebels -o {@emmaIsDead}.
32 emmaGetsSick : type = emma * emmaIsDespaired -o {@debt * @debt * @debt * @debt * !charlesIsConcerned *
       @emma}.
33 emmaLearnsBovaryFatherDeath : type = emma * leonEmmaTogether * charlesIsConcerned * homais -o {@arsenic
        * @inheritance * @leonEmmaTogether * @emma}.
34 emmasLoveForLeonFalters : type = emma * leonEmmaTogether * emmaInLove -o {@emmaIsBored * @emma *
       @leonEmmaTogether}.
35 emmaContractsDebts : type = emma * emmaIsBored -o {@debt * @emma}.
36 emmaCommitsSuicide : type = emma * ruin * arsenic * emmaRebels -o {@emmaIsDead}.
37 init : type =
38  { convent * @emma * @leonIsBored * !charles * !rodolphePastLoveLife * !homais
39         * @emmaSpendsYearsInConvent
40         * @emmaReadsRomanticNovels
41         * @emmaReadsRomanticNovels
42         * @emmaInvitedToBall
43         * @emmaMarriesCharles
44         * @(emmaGoesToBall & emmaDoesNotGoToBall)
45         <.......>
46         * @emmaLearnsBovaryFatherDeath
47          * @emmasLoveForLeonFalters
48         * !emmaContractsDebts
49         * !emmaContractsImportantDebts
50         * @emmaBecomesRuined
51         * @emmaCommitsSuicide
52 }.
53
54 #query * * * 100 (init -o {emmaIsDead}).
```

Fig. 3. Celf Code excerpt of a narrative description inspired by a fragment of the novel *Madame Bovary* [5]. Atomic types corresponding to narrative resources are followed by types describing narrative actions, then the initial environment declaration and finally by the query of 100 attempts to generate stories ending by Emma's death. (The complete file contains 105 lines of code.)

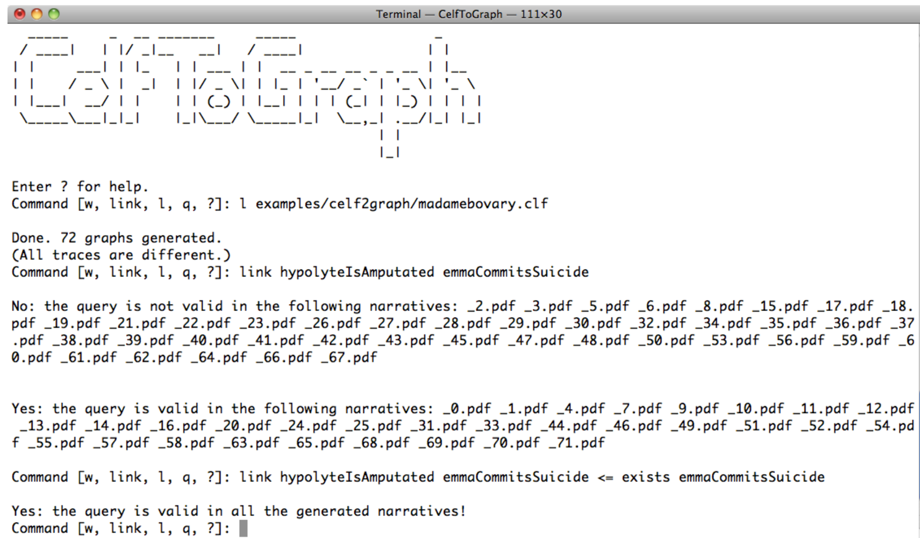

Fig. 4. Loading a Celf file and querying the set of generated plots. The first query discriminates stories depending on a causal path between Emma's suicide by Arsenic and Hypolyte's amputation. The second story restricts the search of the same causal link to stories where Emma takes Arsenic, evidencing Hypolyte's amputation is then one of the cause of Emma's death.

```
emmaMarriesCharles  : type = emma * escapism * grace * charles —o
                                        {emmaIsBored * @emma *!emmaCharlesMarried}
```

Immutable facts and hard rules are modelled as persistent: this is how we represent Emma and Charles married status. We declare the state representing Emma's boredom as linear, since one of the driving force for her actions in the story is to escape boredom. The resources corresponding to Emma and Charles are respectively declared in the initial environment as affine and persistent, because Emma may die in the story and Charles is a constant presence in the modelled fragment (line 38 of Figure 3). This is why the resource corresponding to Emma needs to be preserved by this narrative action. Note that because in this specific code example we are searching for stories where Emma dies, we could have used a linear resource as well.

In addition to the author's notes [6] for filtering through story events irrelevant for the modelled narrative structure, we proceed iteratively, and lazily model a new resource when we model a narrative action involving it. The narrative action corresponding to Emma taking arsenic to poison herself illustrates this process: Emma learns about her father's death from Homais (returning late from a date with Leon) because Charles is afraid to upset her. She learns about inheritance. We first model:

```
emmaLearnsBovaryFatherDeath : type = emma * leonEmmaTogether * charlesIsConcerned * homais —o
                {@inheritance * @leonEmmaTogether * @emma}
```

During the same event, a side conversation occurs between two of the characters present during which Emma incidentally learns where to find arsenic. The importance of this knowledge becomes only apparent when we model the narrative action corresponding to Emma's death. We then modify the code:

```
emmaLearnsBovaryFatherDeath : type = emma * leonEmmaTogether * charlesIsConcerned * homais -o
              {@arsenic * @inheritance * @leonEmmaTogether * @emma}.
emmaCommitsSuicide : type = emma * ruin * arsenic * emmaRebels -o {@emmaIsDead}.
```

Mutually exclusive narrative actions can be explicitly suggested using the choice connective `&` in the declaration of the initial conditions. These can be used to encode key turning points in the narrative that are broadly recognized as such, which is frequently the case when using existing stories as a baseline. We use this connective to model Emma's choice to attend the ball (see line 44 in Figure 3). The use of the choice connective `&` causes variation in the outputs. However, the main mechanism for varied outputs remains the competition for the consumption of resources by different narrative actions.

5.2 Incremental Formalization of the Narrative

As the examples above demonstrate, an advantage of modelling narratives using a programming language is the ability to iteratively fine tune the model. Indeed, programming is an iterative activity alternating between coding and testing phases, and the tool that we developed improves immensely the effectiveness of the testing phase.

Testing can exhibit plots with specific characteristics, as illustrated in Figure 4. We can also verify if the generation has a varied output (differing significantly from the original plot) by exhibiting corresponding plots such as on Figure 5. One can also test the impact of more **narrative drive** on the generation if enforcing an action is desired: by making `emmaAcceptsLeonsAdvances` linear, one can observe the effect on the number and variability of the stories generated (such a modification would generate a smaller number of stories per 100 queries, and all of them would end by Emma's death by poisoning).

Such fine-tuning can help setting up threshold values where levels or a certain amount of a given resource is needed to trigger various specific narrative actions.

5.3 Generated Plots

The complete code corresponding to the extract shown in Figure 3 consists of a total of 105 lines of code, including 31 narrative action descriptions (the rest of the code being mainly atomic declarations). As we have only explicitly encoded one branching choice, the variety of outputs is due to the narrative actions semantics (narrative action's competition for resources, and the fact that different actions can create the resource consumed by another one), and to the forward chaining variability.

The code described allows to **generate** 72 different narrative sequences for 100 attempts. After a comparison of the corresponding plots using the *CelfToGraph*'s command `stats`, we can exhibit 41 different plots (characterised by

(a) Emma does not attend the Vicomte's Ball but still wants to escape her life. She defenestrates when left by Rodolphe.

(b) As in the original story, Emma attends the Vicomte's Ball. Following Rodolphe's departure she becomes sick but later start another liaison with Leon. After being ruined, she ingests Arsenic.

Fig. 5. Two of 41 causally structured plots generated by the code shown in Figure 3. These plots have been exhibited using *CelfToGraph* querying language on the generated set. Nodes in Grey are final action which do not have consequences.

different generated causal structures), meaning that a number of different narrative sequences share the same causal structures. This allows the characterisation of classes of true **story variants**. Figure 5 exhibits examples of story variants among those generated, which have been exhibited by the tool: the first tells a story where Emma jumps through the window following the departure of Rodolphe, and the other a story where she takes arsenic. If we look at the code Figure 3 l.31 32, two narrative actions `emmaJumpsThroughWindow` and `emmaGetsSick` consume the resource `emmaIsDespaired`. When the first is triggered by the forward chaining mechanism, we obtain a story ending with Emma jumping through the window. When requesting 1000 query attempts, we obtain 747 solutions, among which 697 are different narrative sequences, and 226 are different plots (i.e., 226 true story variants).

6 Conclusion

There has been much interest in the use of Linear Logic to represent natural language semantics and the semantics of action and change. Narrative structures are based on the integration of the above phenomena, and Linear Logic programming provides a direct mechanism to operationalize these descriptions

Our first results reported here are clearly encouraging and compare favourably to other narrative generation techniques based on planning, while preserving all the benefits of a declarative representation. This opens perspectives for applications such as Interactive Storytelling, where narrative generation is a default interaction paradigm, allowing narratives to adapt to changes in the environment. In future work, we intend to develop this approach with the definition of an interaction paradigm using Linear Logic's choice connectives and on-the-fly environment modifications. Another interesting line of inquiry would be to explore the possible definition of normal forms for stories generated.

References

1. Girard, J.Y.: Linear Logic. Theoretical Computer Science **50**(1) (1987) 1–102
2. Bosser, A.G., Cavazza, M., Champagnat, R.: Linear Logic for non-linear storytelling. In: ECAI 2010. Volume 215 of Frontiers in Artificial Intelligence and Applications., IOS Press (2010)
3. Girard, J.Y., Lafont, Y.: Linear Logic and lazy computation. In Ehrig, H., Kowalski, R., Levi, G., Montanari, U., eds.: TAPSOFT '87. Volume 250 of LNCS. Springer Berlin / Heidelberg (1987) 52–66
4. Schack-Nielsen, A., Schürmann, C.: Celf — a logical framework for deductive and concurrent systems (system description). In: Automated Reasoning. Springer (2008) 320–326
5. Flaubert, G.: Madame Bovary. Folio (2001) In French. Also available in English, OUP Oxford, reissue edition, 2008.
6. Flaubert, G.: Plans et Scénarios de Madame Bovary. Zuma, Cadeilhan (1995) In French. Presentation, transcription and notes by Yvan Leclerc.

7. Schank, R., Abelson, R.: Scripts, plans, goals and understanding: An inquiry into human knowledge structures. Psychology Press (1977)
8. Miller, R., Shanahan, M.: Narratives in the Situation Calculus. Journal of Logic and Computation **4** (1994) 513–530
9. Kakas, A., Miller, R.: A simple declarative language for describing narratives with actions. Journal of Logic Programming **31** (1997) 157–200
10. Reiter, R.: Narratives as programs. In: KR, Morgan Kaufmann Publishers (2000)
11. Baral, C., Gabaldon, A., Provetti, A.: Formalizing narratives using nested circumscription. Artificial Intelligence **104**(1-2) (1998) 107–164
12. Young, R.M.: Notes on the use of plan structures in the creation of interactive plot. In: Narrative Intelligence: Papers from the AAAI Fall Symposium, AAAI Press (1999)
13. Porteous, J., Cavazza, M., Charles, F.: Applying planning to interactive storytelling: Narrative control using state constraints. ACM Trans. Intell. Syst. Technol. **1**(2) (December 2010) 10:1–10:21
14. Alexiev, V.: The Event Calculus as a Linear Logic program. Technical Report TR95-24, Department of Computer Science, University of Alberta (1995)
15. Masseron, M.: Generating plans in Linear Logic: II. A geometry of conjunctive actions. Theoretical Computer Science **113**(2) (1993) 371–375
16. Cabalar, P.: Causal logic programming. In Erdem, E., Lee, J., Lierler, Y., Pearce, D., eds.: Correct Reasoning. Volume 7265 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 102–116
17. Schroeder, M.: How to tell a logical story. In: Narrative Intelligence: Papers from the AAAI Fall Symposium, AAAI Press (1999)
18. Lang, R.R.: A declarative model for simple narratives. In: Narrative Intelligence: Papers from the AAAI Fall Symposium, AAAI Press (1999)
19. Collé, F., Champagnat, R., Prigent, A.: Scenario analysis based on Linear Logic. In: ACM SIGCHI Advances in Computer Entertainment Technology (ACE), ACM press (2005)
20. Dang, K.D.: Aide à la Réalisation de Systèmes de Pilotage de Narration Interactive: Validation d'un Scénario Basée sur un Modèle en Logique Linéaire. PhD thesis, Université de La Rochelle (2013)
21. Bosser, A.G., Courtieu, P., Forest, J., Cavazza, M.: Structural analysis of narratives with the Coq proof assistant. In: ITP. (2011)
22. Hodas, J.S., Miller, D.: Logic programming in a fragment of Intuitionistic Linear Logic. Information and Computation **110**(2) (1994) 327–365
23. J.Pym, D., Harland, J.A.: A uniform proof-theoretic investigation of Linear Logic programming. Journal of Logic and Computation **4**(2) (1994) 175–207
24. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent Linear Logic programming. In: Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. (2005)
25. Miller, D.: Overview of Linear Logic programming. Linear Logic in Computer Science **316** (2004) 119–150
26. Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University (2002)
27. Andreoli, J.: Logic programming with focusing proofs in Linear Logic. Journal of Logic and Computation **2** (1992) 297–347
28. Greenland, S., Pearl, J., Robins, J.: Causal diagrams for epidemiologic research. Epidemiology (1999) 37–48

A Appendix

This appendix is only for reviewing purposes. Not to appear in the final version. We use it to show the complete Celf code that was shown partially in Figure 3. The file is also available online at:

https://github.com/jff/TeLLer/blob/master/examples/celf2graph/LPNMR13-madamebovary.clf.

```
1  %% Encoding of the Madame Bovary story.
2
3  %% All atoms in the story
4  emma : type.
5  charles : type.
6  homais : type.
7  leon : type.
8  rodolphe : type.
9  emmaCharlesMarried : type.
10 covent : type.
11 novels : type.
12 grace : type.
13 education : type.
14 escapism : type.
15 emmaIsBored : type.
16 ball : type.
17 leonIsBored : type.
18 leonInlove : type.
19 leonInfatuated : type.
20 emmaBelievesLeonsLove : type.
21 rodolpheDecidedToSeduce : type.
22 rodolphePastLoveLife : type.
23 rodolpheEmmaTogether : type.
24 rodolpheIsBored : type.
25 emmaFeelsGuilty : type.
26 hypolyteSurgeryPlans : type.
27 emmaCharlesHappy : type.
28 emmaRebels : type.
29 lheureux : type.
30 debt : type.
31 gift : type.
32 rodolpheIsHumiliated : type.
33 emmaIsDespaired : type.
34 charlesIsConcerned : type.
35 emmaInLove : type.
36 leonEmmaTogether : type.
37 arsenic : type.
38 inheritance : type.
```

```
39 denounced : type.
40 ruin : type.
41 emmaIsDead : type.
42
43 emmaSpendsYearsInCovent : type = emma * covent —o {!novels * !grace * !
       education * @emma}.
44 emmaReadsRomanticNovels : type = emma * novels —o {@escapism * @escapism
        * @emma}.
45 emmaMarriesCharles  : type = emma * escapism * grace * charles —o {
       emmaIsBored * @emma *!emmaCharlesMarried}.
46 emmaInvitedToBall : type = emma * emmaCharlesMarried * grace —o {@ball *
        @emma}.
47 emmaGoesToBall : type = emma * ball * escapism —o {@escapism * @escapism
        * @escapism * @escapism * @emma}.
48 emmaDoesNotGoToBall : type = emma * ball —o {emmaIsBored * @emma}.
49 leonFallsInLove : type = emma * grace * novels * leonIsBored —o {
       @leonInlove * @emma}. % debanged
50 emmaDiscoversLeonsLove : type = emma * leonInlove * emmaIsBored —o {
       @emmaBelievesLeonsLove * @leonInlove * @emma}. % possibility
       insertion what—if here
51 emmaPushesLeonAway : type = emma * leonInlove * emmaBelievesLeonsLove *
       emmaCharlesMarried —o {emmaIsBored * @escapism * @emma}.
52 rodolpheDecidesToSeduceEmma : type = emma * grace * emmaCharlesMarried —
       o {@rodolpheDecidedToSeduce * @emma}.
53 emmaAcceptsRodolpheAdvances : type = emma * escapism * escapism *
       escapism * escapism * rodolpheDecidedToSeduce —o {
       @rodolpheEmmaTogether * @emma}. % debanged
54 rodolpheRelationshipFalters : type =  emma * rodolpheEmmaTogether —o {
       @rodolpheIsBored * @emmaFeelsGuilty * @rodolpheEmmaTogether * @emma
       }. % consume repro
55 charlesDecidesToOperateHypolyte : type = emma * charles *
       emmaCharlesMarried * emmaFeelsGuilty * homais —o {
       @hypolyteSurgeryPlans * @emmaCharlesHappy * @emma}.
56 hypolyteIsAmputated : type = emma * charles * emmaCharlesHappy *
       hypolyteSurgeryPlans —o {!emmaRebels * !lheureux * @emma}.
57 emmaPurchasesGift : type = emma * rodolpheEmmaTogether * lheureux —o {
       @debt * @gift * @emma * @rodolpheEmmaTogether}.
58 emmaOffersGift : type = emma *  rodolpheEmmaTogether * gift —o {
       @rodolpheIsHumiliated * @emma * @rodolpheEmmaTogether}.
59 emmaPurchasesProstheticLeg : type = emma * emmaRebels * lheureux —o {
       @debt * @debt * @emma}.
60 rodolpheBreaksUp : type = emma * rodolpheIsBored * rodolpheIsHumiliated
       * rodolphePastLoveLife * rodolpheEmmaTogether —o {@emmaIsDespaired *
        @emma}.
```

```
61 emmaJumpsThroughWindow : type = emma * emmaIsDespaired * emmaRebels -o {
       @emmaIsDead}.
62 emmaGetsSick : type = emma * emmaIsDespaired -o {@debt * @debt * @debt *
        @debt * !charlesIsConcerned * @emma}.
63 emmaMeetsLeon : type = emma * charlesIsConcerned * homais * grace *
       education -o {@leonInfatuated * @emmaInLove * @emma}.
64 emmaAcceptsLeonsAdvances : type = emma * leonInfatuated * emmaInLove *
       emmaRebels -o {@leonEmmaTogether * @emmaInLove * @leonInfatuated *
       @emma}.
65 emmaLearnsBovaryFatherDeath : type = emma * leonEmmaTogether *
       charlesIsConcerned * homais -o {@arsenic * @inheritance *
       @leonEmmaTogether * @emma}. % consume repro
66 emmaReimbursesSomeDebt : type = emma * inheritance * lheureux * debt *
       debt * debt * debt -o {@emma}.
67 leonsMotherReceivesAnonymousLetter : type = emma * emmaCharlesMarried *
       leonEmmaTogether -o {@denounced * @leonEmmaTogether * @emma}.
68 emmasLoveForLeonFalters : type = emma * leonEmmaTogether * emmaInLove -o
        {@emmaIsBored * @emma * @leonEmmaTogether}.
69 bossDiscussesEmmaWithLeon : type = emma * leonEmmaTogether *
       leonInfatuated * denounced -o {@emma * @leonEmmaTogether}.
70 emmaContractsDebts : type = emma * emmaIsBored -o {@debt * @emma}.
71 emmaContractsImportantDebts : type = emma * emmaIsBored -o {@debt *
       @debt * @debt * @emma}.
72 emmaBecomesRuined : type = emma * debt * debt * debt * debt * debt *
       debt * debt -o {@ruin * @emma}.
73 emmaCommitsSuicide : type = emma * ruin * arsenic * emmaRebels -o {
       @emmaIsDead}.
74 init : type =
75  { covent * @emma * @leonIsBored * !charles * !rodolphePastLoveLife * !
        homais
76           * @emmaSpendsYearsInCovent
77           * @emmaReadsRomanticNovels
78           * @emmaReadsRomanticNovels
79           * @emmaInvitedToBall
80           * @emmaMarriesCharles
81           * @(emmaGoesToBall & emmaDoesNotGoToBall)
82           * @leonFallsInLove
83           * @emmaDiscoversLeonsLove
84           * @emmaPushesLeonAway
85           * @rodolpheDecidesToSeduceEmma
86           * @emmaAcceptsRodolpheAdvances
87           * @rodolpheRelationshipFalters
88           * @charlesDecidesToOperateHypolyte
89           * @hypolyteIsAmputated
90           * @emmaPurchasesGift
```

```
91          * @emmaOffersGift
92          * @emmaPurchasesProstheticLeg
93          * @rodolpheBreaksUp
94          * @emmaJumpsThroughWindow
95          * @emmaGetsSick
96          * @emmaMeetsLeon
97          * @emmaAcceptsLeonsAdvances
98          * @emmaLearnsBovaryFatherDeath
99          * @emmaReimbursesSomeDebt
100         * @leonsMotherReceivesAnonymousLetter
101         * @emmasLoveForLeonFalters
102         * @bossDiscussesEmmaWithLeon
103         * !emmaContractsDebts
104         * !emmaContractsImportantDebts
105         * @emmaBecomesRuined
106         * @emmaCommitsSuicide
107 }.
108
109 #query * * * 100 (init -o {emmaIsDead}).
```