PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 07/17

Network Traversal as an Aid to Plot Analysis and Composition

Edirlei S. de Lima
Vinicius M. Gottin
Antonio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 07/17 ISSN 0103-9741
Editor: Prof. Carlos José Pereira de Lucena May, 2017

Network Traversal as an Aid to Plot Analysis and Composition

Edirlei S. de Lima[1]
Vinicius M. Gottin[2]
Antonio L. Furtado[2]

[1] UERJ/IPRJ, Departamento de Modelagem Computacional, Nova Friburgo, Brasil
[2] PUC-Rio, Departamento de Informática, Rio de Janeiro, Brasil
edirlei@iprj.uerj.br, {vgottin,furtado}@inf.puc-rio.br

Abstract: We claim that by combining a chosen set of variants of the same narrative pattern into a network structure, an analysis of the pattern can be achieved that is revealing enough to provide indications on how to help non-professional writers to interactively compose new plots suitable to their tastes. To find the problems entailed by adopting this strategy, and to test the adequacy of our decisions, we developed a prototype tool and applied it to a set of four radically distinct variants of the popular Little Red Riding Hood folktale.

Keywords: Narrative Genre, Folktale Types, Computational Narratology, Network Structure, Plot analysis, Interactive Plot Composition, Logic Programming.

Resumo: Afirmamos que, combinando um conjunto seleto de variantes do mesmo padrão narrativo em uma estrutura de rede, torna-se possível efetuar uma análise desse padrão que fornece indicações sobre como ajudar escritores não-profissionais a compor novos enredos a seu gosto. Com o objetivo de detetar os problemas decorrentes da adoção desta estratégia, e para testar a eficácia de nossas decisões, desenvolvemos um protótipo e o aplicamos a quatro variantes radicalmente distintas do popular conto folclórico de Chapeuzinho Vermelho.

Palavras-chave: Gêneros Narrativos, Tipos Folclóricos, Narratologia Computacional, Estrutura de Rede, Análise de Enredos, Composição Interativa de Enredos, Programação em Lógica

In charge of publications

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br

1. Introduction

The present work aims, as a preliminary objective, at the analysis of *operationally-defined narrative sub-genres* extracted from literary genres that obey some fixed set of conventions, and next, on the basis of this analysis, at the interactive composition of new narratives by non-professional writers. To pursue these two complementary goals, we shall introduce along this paper a modelling strategy, which involves representing the intended sub-genre by combining a chosen repertoire of distinct versions (*variants*) of the same narrative pattern into a *network structure*.

Our work, following a *computational narratology* orientation [12], was very much influenced by widely-known literary studies. The first is due to Mieke Bal, who distinguished in her book [7] three layers in the process of narrative composition – *fabula*, *story* and *text*. Here we shall focus on the *fabula* layer, at which what occurs in the true or fictional world of the narrative is expressed by *events* and *plots* (understood as sequences of events) characteristic of the given genre.

Secondly, our way to conceptually specify events was inspired on Propp's seminal proposal [34], according to whom the events that can happen in a Russian folktale are limited to those produced by 31 "functions". We soon realized that these folktale-oriented functions (or, in general, the functions appropriate to model other genres equally subject to strict conventions) could be cast into the form of logical terms – which we chose to treat as event-producing operations [15]. We also found that, by defining such operations in terms of their pre-conditions and post-conditions (in consonance with the STRIPS method [19]), one is able to derive the partial-order requirements that well-formed narrative plots pertaining to the chosen genre must obey. A practical advantage of the method is that it can be readily applied to chain events together. For example, if the pre-conditions for performing an operation O_1 can be fulfilled by the post-conditions of another operation O_2, a backward-chaining plan-generation algorithm can articulate an event sequence wherein O_2 precedes O_1, and so on and so forth, recursively. Approaching plot-generation as a plan-generation problem, as we did quite early [16], brings to mind an important aspect of plots: the fact that the required ordering is only partial, plus the fact that more than one operation may have the same main effects, introduce the possibility of finding a variety of different plots (by causing the plan-generation algorithm to backtrack) that lead from a given initial narrative world state to some target state.

Thirdly, an ample illustration of this variety is provided by another proposal to characterize folktale types, namely the *Aarne-Thompson Index* (henceforward simply *Index*) [2,42,43]. Each type corresponds to a basic narrative pattern, which matches innumerable folktales composed along the centuries in different regions of the world. Folktales classified under a given type are called *variants*. The target state to be reached as the narrative concludes does not have to be exactly the same in all variants of a type, a usually small set of possible *outcomes* being contemplated as compatible with the type (some of which – but not necessarily all – appearing as a "happy ending" to the reader).

The *Index* describes a large collection of folktale *types*, such as type **AT 333**[1] **– The Glutton**, which comprises what is widely known as the tale of **Little Red Riding Hood** (henceforward **LRRH**), on which we shall concentrate in this paper. Looking at the *Index* ([2], page 125), one will get the impression that the authors have only considered the

[1] **AT 333** designates **A**arne and **T**hompson's original classification [2], now often written **ATU 333** to reflect **U**ther's revision [43].

brothers Grimm classic variant [21] to define type **AT 333**. Indeed, from the two phases that they identify – *Wolf's Feast* and *Rescue* – only the former is present in Perrault's no less famous variant [33], whose outcome is consequently very much far from a "happy ending".

The thrust of this paper is to show that, if one wishes to consider more than one variant when defining a type, such linear patterns are not adequate, and that a far better strategy is to combine a chosen set of variants into a *network-structured pattern*, where alternative sequences of events can be clearly exposed. Over the network, the consequent operationally-defined sub-genre can be appropriately analyzed, and new variants can be obtained by traversing still untried paths consisting of nodes originating from different variants.

To illustrate our approach we chose, after a brief survey of the literature related to type **AT 333** [21,33,17,10,32,41,26,27], four strikingly divergent variants, which tell the little girl's story with quite different outcomes. We developed a prototype to examine in practice the problems entailed by constructing a network pattern and by generating novel variants, especially the need to impose restrictions, inherent in *blending* processes as pointed out by Fauconnier and Turner [18].

The prototype utilizes logic programming (SWI-Prolog), plus other programming languages to implement auxiliary features. Due to its modular architecture, it is not confined to a single domain. Indeed, we have already applied it to an academic information system, where the "variants" used to build the network where event sequences extracted from a database log. In order to achieve the intended degree of generality, all clauses that are specific to a domain, in the present case the **LRRH** domain, are kept in a separate module.

The rest of this paper is structured as follows. Section 2 describes the general methods employed to build the networks and to compose new plots. Sections 3 and 4 concern the specific features of the **LRRH** example application, section 3 being dedicated to construction and analysis, and section 4 to plot composition. Section 5 presents the architecture of our prototype application and deals with the visual, voice, and template-based textual language generation techniques employed for drawing the network during the plot composition, and used to dramatize the generated stories. Section 6 surveys related work. Section 7 contains some concluding remarks.

2. Combining event sequences into networks

Given an application domain, we shall focus on some of the classes of events that may occur in the domain. Event sequences pertaining to the chosen domain are time-ordered sets of terms of the form `ev(p1, p2, …, pn)`, where each `ev` denotes one such class of events, and the values of the `pi` parameters serve to characterize different instances of the class.

Thus, in a folktale domain, an example of event sequence may be:

```
[ask_to_take('Mother', 'Little Red Riding Hood', 'cake and butter',
 'Grandmother'), go('Little Red Riding Hood', 'the woods'), meet('Little Red
 Riding Hood', 'Wolf'))]
```

involving three classes of events, namely `ask_to_take`, `go` and `meet`.

Here, we shall be concerned with different event sequences that represent entire versions of the same basic story – the so-called story *variants* – classified under a folktale *type*.

Starting from section 3, we shall illustrate our network approach by specifically applying it to type **AT 333** of the Aarne-Thompson *Index* [2], which covers the **Little Red Riding Hood** (**LRRH**) time-honoured and universally popular tale.

A first visual comparison of any set of variants is often enough to notice sub-sequences that look similar, as well as sub-sequences that converge or diverge at some point. The main claim of our paper is that sub-sequence similarities, convergences and divergences can best be characterized by combining the variants into a network structure. Similar sub-sequences can be unified, after suitable adaptation. In turn, employing workflow terminology [11], divergences and convergences can be represented, respectively, by way of *fork* and *join* nodes. Also, the network connectivity properties promptly suggest the possibility of finding new event sequences, by traversing edges originating from two or more variants.

To combine variants of widely different time period, nationality and narrative style (supposing, naturally, that they are all available in the same language, English in the present case) the first step is to apply *generalization* to map similar sub-sequences into some standard format. This preliminary adjustment may involve the name of the events, as well as the names of characters, places and objects that constitute their parameters. Event instances of the same name but different values for some parameters, such as `ev(a,b)` and `ev(a,c)`, can be made identical, if desirable, either by replacing `b` and `c` by a constant that subsumes them both (e.g. by denoting a species of which `b` and `c` are sub-species), or by a variable, or simply by omitting the conflicting parameters and writing `ev(a)`.

To more clearly explain our method of generating a network by combining variants, let us consider the following parameter-less oversimplified sequences:

```
[fa]
[fa, fb]
[fa, fe, fc, fd]
[fc, fd]
```

The first version of the network is constructed by simply adding two special events to all four sequences, called `ini` and `end`, to respectively mark the common origin and the common finishing point of the sequences. Having only in common the nodes associated with these two events, the four sequences are put together in a trivial network format, as otherwise totally separate paths, with different `Ni` node labels associated with the events. In Figure 1 below, `N1` labels the `ini` event, and `N3` labels `end`, and the four sequences correspond to the following paths along the network:

```
[N1:ini,  N2:fa,  N3:end]
[N1:ini,  N4:fa,  N5:fb,  N3:end]
[N1:ini,  N6:fa,  N7:fe,  N8:fc,  N9:fd,  N3:end]
[N1:ini,  N10:fc, N11:fd, N3:end]
```

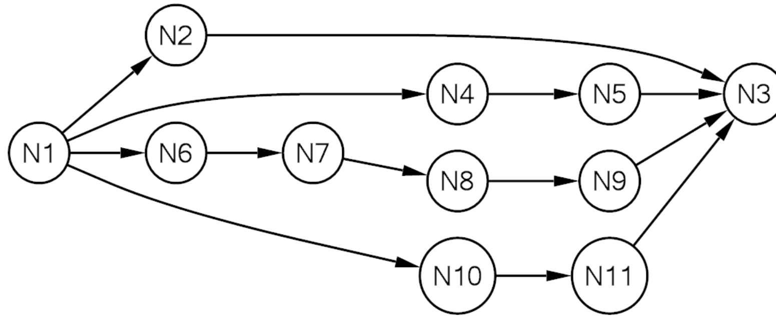

Figure 1 – the first trivial network

Notice that each node appears associated with just one event, which is the last one in the sequence leading to the node. However, this is just a convenient abbreviation of the *node-prefix* format, which we adopted in view of the possibility of multiple occurrences of the same event in an event sequence. The internal representation of a node `Ni` involves the entire series of events, starting from `ini`, whose effect is the *state* holding at `Ni`. For example, the second event sequence is recorded and processed in the form:

```
[N1:[ini], N4:[ini,fa], N5:[ini,fa,fb], N3:[ini,fa,fb,end]]
```

The first step to start combining the sequences involves finding what sub-sequences are identical, and therefore should be represented just once. Thus the identical two-element sub-sequence consisting of `ini` and `fa`, which is repeated in the first three event sequences, corresponding to three edges in the network (`N1-N2, N1-N4, N1-N6`), is represented by a single edge (`N1-N2`) in the modified network of Figure 2. The same happens with the sub-sequences consisting of `fc` and `fd`, which occur in the third and fourth event sequences, respectively.

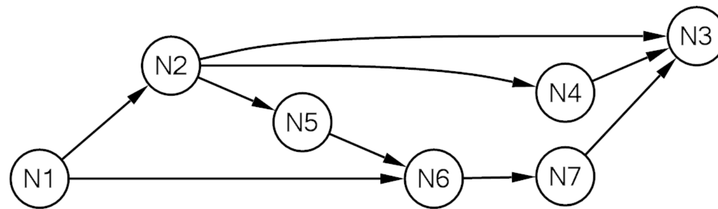

Figure 2 – superposing identical sub-sequences

The four sequences now correspond to the following paths across the network:

```
[N1:ini,  N2:fa,  N3:end]
[N1:ini,  N2:fa,  N4:fb,  N3:end]
[N1:ini,  N2:fa,  N5:fe,  N6:fc,  N7:fd,  N3:end]
[N1:ini,  N6:fc,  N7:fd,  N3:end]
```

where the superposed sub-sequences were `ini-fa`, which occurs in the three first sequences, and `fc-fd`, which is common to the third and fourth sequences.

If a more compact form is desired, one more stage can be added, in order to condense into single nodes any number of events that are different but denote states characterized by

similar (not necessarily equal) situations. Suppose that events fb and fd produce the same main effects. Also suppose that event fe undoes the effect of fa, and thus, if the occurrence of fa leads from N1 to N2, fe would revert to N1 (the ini event), thereby introducing a loop in the network. These considerations motivate the condensation of the nodes N4:fb and N7:fd into a single node N4 encompassing fb and fd, and of nodes N1:ini and N5:fe into a single node N1 encompassing ini and fe. Node labels N5 and N7 vanish, and N6 is relabeled N5 in the final five-node network, represented in Figure 3.

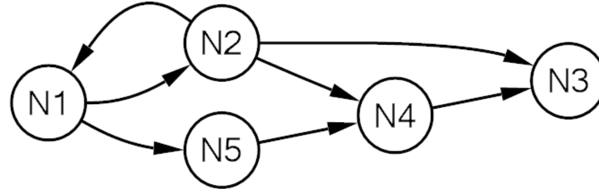

Figure 3 – final condensed network

The four sequences are represented as follows, in this fairly compact network:

```
[N1:ini,  N2:fa,  N3:end]
[N1:ini,  N2:fa,  N4:fb,  N3:end]
[N1:ini,  N2:fa,  N1:fe,  N5:fc,  N4:fd,  N3:end]
[N1:ini,  N5:fc,  N4:fd,  N3:end]
```

As this brief example-based description indicates, our method allows a choice between two possible results when comparing two node-prefixes V1:T1 and V2:T2 taken, respectively, from event-sequences S1 and S2. Let E1 be the last event in T1 and E2 the last event in T2:

- the fuller case: (a) V1 and V2 are unified if E1 and E2 are the same. In Figure 1, N2 and N4 end in fa, and so they are unified as N2 in Figure 2. The same happens with N2 and N6. Also in Figure 1, N8 and N10 both end in fc and are unified in Figure 2 as N6.

- the condensed version: V1 and V2 are unified as in (a), and also if both (b) and (c) hold:
  - (b) E1 and E2 are considered *similar*; where similar events are those explicitly declared as such in a list of *allowed_convergent* pairs
  - (c) V1 and V2 lead to a similar state, that is:
    (c.1) V1:T1 is followed in S1 by V3:T3, with E3 being the last event in T3; and
    (c.2) V2:T2 is followed in S2 by V4:T4, with E4 being the last event in T4; and
    (c.3) E3 and E4 are the same event or are events considered similar.

In the example above, nodes N4 and N7 from Figure 2 are unified as N4 in Figure 3 because the pair fb-fd is on the *allowed_convergent* list and both are followed by nodes representing a similar state – indeed, both are followed by N3 and thus E3=E4=end. Also,

nodes `N5` and `N1` from Figure 2 are unified as `N1` in Figure 3 because the pair `fe-ini` is on the *allowed_convergent* list and both are followed by `N6`, with the consequence that E3=E4=`fc` .

Complementing the general specification of the method, certain domain-dependent clauses must be supplied in a separate module. These include, first of all, the event sequences to be merged to draw the network. The other clauses serve to guide the process, in particular by defining the mappings from the events in the original event sequences into standard generalized formats, and by characterizing the converging cases wherein condensation should take place (with the help of the above mentioned *allowed_convergent* list), etc.

We found that it is necessary to submit all such clauses to fine tuning revision. At that phase, further clauses can be added within an even narrower range, in view of characteristics of the generated network itself. In special, it is most convenient to partition the network into significant components, thus allowing to generate *summary networks*, of great importance towards the analysis of the sub-genre determined by the chosen variants.

After applying enough attention to analysis, it is time to exploit the composition of new stories, or should we say new variants. A guiding feature, analogous to the **GPS** positioning systems that car drivers employ to find the way to their destination, can optionally be activated, whereby the user is prompted to choose among the possible *outcomes*, understood as the concluding events of the summary network. If this feature is active, at each fork node the choices leading to the desired outcome will be recommended. Next, a frame-structured window is made available to users intent on assigning names of their choice to the characters. Next, the user starts the composition process, by trying different choices at the fork nodes and at nodes condensing more than one event.

Here, too, the general path-generating algorithm is conditioned by a number of domain-dependent special clauses, whose purpose is to avoid incongruent situations that may arise when combining sequences taken from different variants. These clauses (as will be illustrated through examples in section 4) limit the choices offered to users at fork or condensed nodes and, if only one option remains, the user is not called to intervene and the generation proceeds without interruption. Any faulty nodes encountered during the application of these clauses are marked as excluded from the generated story.

The reason why such conflicts may occur is that similarity in terms of main effects of events does not mean that secondary effects also fully coincide – hence adaptations need to be done. For the necessity to accomodate potentially conflicting elements, see how the subject of *blending* is creatively exploited by Fauconnier and Turner [18].

A simple numerical evaluation, to assess to what extent a network enables the composition of a diversity of stories, provides a later phase of analysis. As shown in section 4, this is easily done by elementary graph traversal and counting algorithms, which consider both the network resulting from a combination of variants and the associated summary network.

3. Analyzing an LRRH set of variants

Here we shall deal with four variants of the very popular folktale broadly known as *Little Red Riding Hood* (**LRRH** for short). In Aarne-Thompson's *Index*, the story is classified under type **AT 333**, characteristically named **The Glutton**, and is described approximately as follows, noting that the plot comprises two major episodes [2]:

- The wolf or other monster devours human beings until all of them are rescued alive from his belly.
I. *Wolf's Feast*. By masking as mother or grandmother the wolf deceives and devours a little girl whom he meets on his way to her grandmother's.
II. *Rescue*. The wolf is cut open and his victims rescued alive; his belly is sewed full of stones and he drowns, or he jumps to his death.

The first variant that we chose is the classic *Le Petit Chaperon Rouge* (Little Red Riding Hood), composed in France in 1697 by Charles Perrault [33], during the reign of Louis XIVth. It consists of the first episode alone, so that there is no happy ending, contrary to what children normally expect from nursery fairy tales. The little girl, going through the woods to see her grandmother, is accosted by the wolf who reaches the grandmother's house ahead of her. The wolf kills the grandmother and takes her place in bed. When the girl arrives, she is astonished at the "grandmother"'s large, ears, large eyes, etc., until she finally asks about the long teeth, whereat the wolf gobbles her up.

The second, perhaps even more influential classic variant, is that of the brothers Grimm (Jacob and Wilhelm), written in German and entitled *Rotkäppchen* (Little Red Cap) [21], first published in 1812. It encompasses the two episodes. Rescue is effected by a hunter, who finds the wolf sleeping and cuts his belly, allowing the girl and her grandmother to escape. The wolf has his belly filled with heavy stones fetched by the girl, wakes up, tries to run away and falls dead, unable to carry the weight.

Whilst in the classic variants considered so far the girl is presented as naive, in contrast to the clever villain, the situation is reversed in the *Conte de la Mère-grand* (The Story of Grandmother), collected by folklorist Achille Millien in the French province of Nivernais, circa 1870, and later published by Paul Delarue [17]. In this variant, which some scholars believe to be closer to the primitive oral tradition, the villain is a "bzou", a werewolf. After killing and partly devouring the grandmother's body, he stores some of her flesh and fills a bottle with her blood. When the girl comes in, he directs her to eat and drink from these ghastly remains. Then he tells her to undress and lie down on the bed. Whenever the girl asks where to put each piece of clothing, the answer is always: "Throw it in the fire, my child; you don't need it anymore." In the ensuing dialogue about the peculiar physical attributes of the fake grandmother, when the question about her "big mouth" is asked the Bzou gives the conventional reply: "All the better to eat you with, my child!" – but this time the action does not immediately follow the words. What happens instead is that the girl asks permission to go out to relieve herself, which is a ruse whereby she ends up outsmarting the villain and safely going back to her mother's home.

The deviations from the type paradigm apparent in certain texts can be even more extreme. Can the *Uncle Wolf* story [10], for instance, collected by Italo Calvino, still be classified as a variant of type **AT 333: The Glutton**? The trouble is that here the girl is the first to reveal herself as "glutton". She does not resist the temptation to eat and drink all that her mother was sending to Uncle Wolf in return for the loan of a skillet, offering him instead an ugly mess composed of donkey manure, dirty water and lime. He is not deceived and threatens her: "Tonight I'm coming to eat you!" He sneaks into the house, repeatedly announcing where he is at each moment until reaching the girl's room and eating her, in a frightening ghost-like sequence indicative of type **AT 366: The Man from the Gallows**. However we felt justified to include this story as a fourth, somewhat transgressive, variant of the type on hand, since Italo Calvino himself affirms: 'This extremely simple type – and I followed one of the richest versions – will lead to the perfect grace of "Little Red Riding Hood"'[10, pp. 725-6].

The event sequences corresponding to the four **LRRH** variants are shown in Appendix 1. The events are expressed by terms of the form `ev(p1, p2, …, pn),` as mentioned in section 2.

Several event-mapping rules are applied to allow the generalization process that will result in the network structure. For example, the different gifts that the girl brings to her grandmother at her mother's behest, "cake and butter" in Perrault and "cake and wine" in Grimm, are replaced by "food" in both the `ask_to_take` and the `deliver` events. The girl herself is differently called "Little Red Riding Hood", "Little Red Cap", and "Little Girl" – since no real difference is implied by this diversity of names, we chose the last name to designate her in all events. By contrast, we preserved the three surely distinct characterizations of the villain, as "Wolf", "Bzou", and "Uncle Wolf". The `go(X,Y)` event, which in general should specify both the agent X and the place Y, is renamed and reduced to a single parameter, `enter(X)` when the agent X is the "Hunter", so that this character will be able to succour the victims no matter where the villain has attacked them. For convenience (cf. the next paragraph), the event `deliver` is renamed `deliver_false`, whenever performed with a misleading purpose.

Among the rules allowing condensation in a single node of events with similar main effects, a typical example is the recognition of the two events `fool(X,Y)` and `deliver_false(X,F,Y)`, where X is the victim, F is some adulterated food or beverage, and Y is the villain, as alternative tactics employed by the victim to deceive the villain.

Starting to analyze the resulting network (shown in section 5, at the bottom partition of Figure 6), we found that it comprised more than just the two phases ('Wolf's Feast' and 'Rescue') proposed by the *Index*, which, as we pointed out before, seems to have exclusively considered Grimm's narrative. We detected 9 phases, which we duly registered together with the node labels corresponding to the events that marked the sub-paths involved: 'Preparation', 'Villainy against Grandmother', 'Girl's gluttony', 'Girl as cannibal', 'Girl fools villain', 'Villainy against Girl', 'Girl suffers retaliation', 'Safe return home', 'Rescue'.

Having identified these phases, we were able to:

1. summarize each of the variants
2. draw a summary network
3. identify the possible outcomes

Summarizing the four variants, in view of the identified phases, yields the following event sequences:

```
[Preparation, Villainy against Grandmother, Villainy against Girl]
[Preparation, Villainy against Grandmother, Villainy against Girl,
 Rescue]
[Preparation, Villainy against Grandmother, Girl as cannibal,
 Girl fools villain, Safe return home]
[Preparation, Girl's gluttony, Girl fools villain,
 Girl suffers retaliation]
```

which, in turn, can be combined into the network structure of Figure 4.

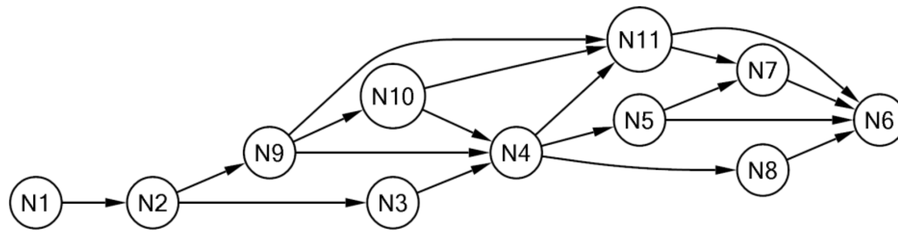

Figure 4 – Summary network

The intended meaning of the network is expressed, as before, by associating the node labels to the last event in the sub-sequence (prefix) leading to each node:

```
N1:[ini]
N2:[Preparation]
N3:[Girl's gluttony]
N4:[Girl fools villain]
N5:[Girl suffers retaliation]
N6:[end]
N7:[Rescue]
N8:[Safe return home]
N9:[Villainy against Grandmother]
N10:[Girl as cannibal]
N11:[Villainy against Girl]
```

The only possible outcomes of the **LRRH** stories are, naturally, the last events in the event sequences wherefrom the network was generated, namely 'Villainy against Girl', 'Rescue', 'Safe return home', 'Girl suffers retaliation'. In the network, these are the events associated with the nodes adjacent to N6 (the `end` node), whose labels are, respectively, N11, N7, N8, N5.

4. Composing new LRRH plots

As indicated in section 2, the user can call the *plot-generation program* to compose new plots by traversing the network resulting from the four selected variants, making choices at fork nodes, as well as at nodes wherein a plurality of similar events are condensed. To reach their decisions, users can optionally summon the help of the **GPS** facility, which begins by asking which of the four outcomes should be aimed at as final destination of the path-finding effort. Based on that, the program recommends, at each decision point, what choices would lead to the indicated outcome.

Another preliminary decision refers to the (also optional) assignment of proper names to the five members of the cast, playing the roles of Victim, Villain, Mother, Grandmother, and Hunter.

The type of villain – Wolf, Bzou (werewolf), or Uncle Wolf – is determined at an early stage by the plot-generation program, specifically when, positioned at fork node N4 of the network, the user chooses to proceed towards either N5, or N22, or N32, whereby a `meet` event is added to the story having as argument one of those three possibilities.

The user is free to accept or ignore the (optional) recommendations of **GPS**. He can even assume a playful attitude, choosing randomly and waiting to see what story might

emerge. If, on the contrary, his preference is not to leave the developments to chance, two features are available to promote better informed decisions. Firstly, the network displayed by the interface has a *tooltip* property whereby the event(s) associated with the node over which the cursor is placed are revealed. Secondly, before coming to a decision, the user can learn about the consequences of the alternative choices by typing, before a definite reply, either '?', in which case the concluding summary events (outcomes) resulting of each choice are listed, or '??', which lists the entire summaries.

While plot-generation proceeds, the network is redrawn on the screen after each decision stage, to mark the path thus far traversed by way of equally coloured nodes and edges. Another colour signals the immediately adjacent nodes, one of which will next be linked to reflect the user's choice.

As alerted in section 2, plot-generation across a network that combines a diversity of variants is prone to occasional incongruences, which must be the object of different kinds of *restrictions*.

Suppose, for example, that in the path along which the story is generated only the girl, not her grandmother, is gobbled by the villain, and also suppose that, further on, the villain's belly is cut by the hunter. Now, in continuation, two `jump_out_of` events occur in the resulting path, one enacted by the victim and the other by the grandmother – but the latter is physically impossible, given the obvious rule that one can be disgorged only if previously devoured. It is also justifiable (though, in this case, for plausibility rather than physical impossibility) that the events `ask_to_take` and `deliver` imply each other – and yet there are cases where one of these events does occur while the other is absent. Our solution to these cases meets two requirements: the offending events are removed from the story, but the respective nodes are kept in the linked network path – with a distinguishing gray colour – so as to avoid the impression of a path discontinuity.

Another kind of restriction refers to the set of events associated with condensed nodes, exactly one of which has to be chosen by the user. For example, the victim has two ways to deceive the villain, corresponding to the events `fool` and `deliver_false`. In this particular case, the choice is predetermined by a test to be performed immediately before the decision point: a `deliver_false(X,Y,Z)` event must be chosen if and only if a `make(X,Y)` event has previously occurred. The restricted choices are excluded from the set of options submitted to the user and, since just one option remains, the decision point is simply skipped by the plot-generation process, which advances after automatically incorporating the one remaining event.

This kind of simplification is also applicable to fork nodes, even though we had no opportunity to employ it in the context of the **LRRH** variants. To give an example, taken from an academic domain, at a fork node leading to different `pass` events a restriction rule checks whether any event `drop(X,Y)` (student X has dropped course Y) has previously occurred, in which case the corresponding `pass(X,Y)` event (student X has obtained a passing grade in course Y) is removed from the set of options.

To give an example of interactive plot generation, which incidentally is the same that is represented in Figure 6, suppose the user takes the following options:

```
invokes the GPS feature
indicates as outcome: rescue
names the cast: victim: Floriel, villain: Lupin, mother: Meg,
                grandmother: Queen Mab, hunter: Eric the Huntsman
chooses at fork nodes: N2,N32,N48,N16 - the last two recommended by
                       GPS as the only adequate ones
chooses an event at node N4, namely: go(Floriel, the crossroad)
```

The results are:

Plot:
```
[N1,N2,N4,N32,N33,N34,N35,N36,N37,N38,N39,N40,N41,N42,N43,N44,N45,
 N46,N47,N29,N30,N48,N14,N16,N17,N18,N19,N21,N15]
```

Events in the plot:
```
N1:ini
N2:give(Queen Mab,red covering,Floriel)
N4:go(Floriel,the crossroad)
N32:meet(Floriel,Lupin)
N33:ask(Floriel,skillet,Lupin)
N34:give(Lupin,skillet,Floriel)
N35:make(Meg,pancakes)
N36:ask_to_take(Meg,Floriel,pancakes,Lupin)
N37:ask_to_take(Meg,Floriel,bread,Lupin)
N38:ask_to_take(Meg,Floriel,wine,Lupin)
N39:ingest(Floriel,pancakes)
N40:ingest(Floriel,bread)
N41:ingest(Floriel,wine)
N42:make(Floriel,false pancakes)
N43:make(Floriel,false bread)
N44:make(Floriel,false wine)
N45:go(Floriel,Lupin's house)
N46:deliver_false(Floriel,false pancakes,Lupin)
N47:deliver_false(Floriel,false bread,Lupin)
N29:deliver_false(Floriel,false wine,Lupin)
N30:go(Floriel,Meg's house)
N48:go(Lupin,Meg's house)
N14:eat(Lupin,Floriel)
N16:sleep(Lupin)
N17:enter(Eric the Huntsman,Meg's house)
N18:cut(Eric the Huntsman,Lupin,axe)
N19:jump_out_of(Floriel,Lupin)
N21:die(Lupin)
N15:end
```

Remark: in the network shown at the bottom partition of the user interface (Figure 6), path nodes N3 and N20 are shown in gray, since, although they belong to the path traversed to generate the plot, they were excluded from the plot due to restrictions described in this section:

```
N3:ask_to_take(Meg,food,Quenn Mab)
N20:jump_out_of(Queen Mab,Lupin)
```

Additional information displayed:
```
*** The villain Lupin is of type Uncle Wolf
*** Summary:
[Preparation, Girl's gluttony, Girl fools villain,
 Girl suffers retaliation, Rescue]
```

The total number of plots that can be thus generated is 378, but this number should not be taken as an indicator of the variety of possibilities offered by the method. A better, more conservative indicator can be found by first applying the same simple path-traversing algorithm to the summary network to determine the number of different summaries, which turns out to be just 13, and then proceed to group the plots according to their summaries. The 13 different summaries and the number of plots that correspond to each of them, totaling 378 as anticipated, are listed below:

[Preparation,Girl's gluttony,Girl fools villain,Girl suffers retaliation] - **18**
[Preparation,Girl's gluttony,Girl fools villain,Girl suffers retaliation,Rescue] - **18**
[Preparation,Girl's gluttony,Girl fools villain,Safe return home] - **18**
[Preparation,Villainy against Grandmother,Girl as cannibal,Girl fools villain,Safe return home] - **36**
[Preparation,Villainy against Grandmother,Girl as cannibal,Girl fools villain,Villainy against Girl] - **36**
[Preparation,Villainy against Grandmother,Girl as cannibal,Girl fools villain,Villainy against Girl,Rescue] - **36**
[Preparation,Villainy against Grandmother,Girl as cannibal,Villainy against Girl] - **18**
[Preparation,Villainy against Grandmother,Girl as cannibal,Villainy against Girl,Rescue] - **18**
[Preparation,Villainy against Grandmother,Girl fools villain,Safe return home] - **36**
[Preparation,Villainy against Grandmother,Girl fools villain,Villainy against Girl] - **36**
[Preparation,Villainy against Grandmother,Girl fools villain,Villainy against Girl,Rescue] - **36**
[Preparation,Villainy against Grandmother,Villainy against Girl] - **36**
[Preparation,Villainy against Grandmother,Villainy against Girl,Rescue] - **36**

One may additionally want to consider a grouping by different outcomes, which are 4 as explained at the end of section 3:

Girl suffers retaliation - **18**
Rescue - **144**
Safe return home - **90**
Villainy against Girl - **126**

5. An overview of the prototype architecture

The architecture of our prototype application is composed of five different modules (Figure 5). The SWI-Prolog Interface is responsible for creating the network structure by combining variants defined in the Domain Database. After being generated, the network structure is sent to the Draw Network module, which is responsible for visually displaying the network on the screen. The SWI-Prolog Interface also implements the path-generation algorithm, which allows users to compose new story variants with or without the assistance of the **GPS** feature. Once a new story variant is created, it is sent to the Plot Viewer module, which generates and displays the story events as a comics-based story-board, with image and text for each event. If requested by the user, the SWI-Prolog Interface also generates and preserves in a text file, under a chosen story title, a more articulate (in the sense to be explained in sub-section 5.3) textual description of the entire story just produced. This facility allows the user to keep a personal library, whose pieces can be sent at any time to the Storyteller module for vocal narration.

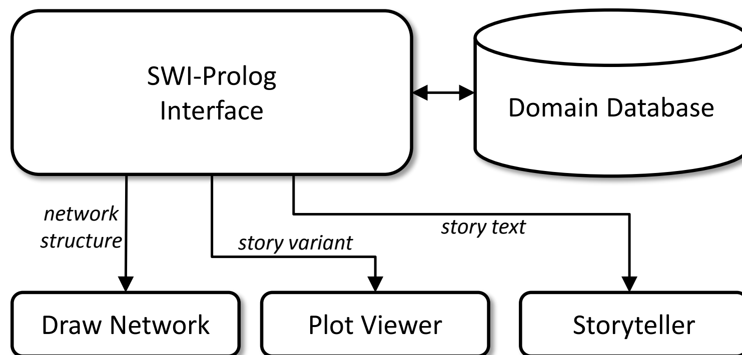

Figure 5 – System architecture

The final user interface, as shown in Figure 6, begins by generating and displaying the network in the lower partition of the screen (Draw Network module), and listing on the upper left partition the event sequences of the variants employed (SWI-Prolog Interface), with each event preceded by the corresponding network node determined by the program. On the same partition the user can then start creating his plot, making decisions at each stage, and watching the resulting path as it is obediently drawn, in colours, over the network. As soon as the plot is ready, a comics-based story-board is shown on the upper right partition (Plot Viewer module). The user can additionally want to listen to a vocal telling, and can order a copy of the narrative on a TXT file, named with the story-title of his choice, from which he may, at any time, hear again the vocal telling (Storyteller module).

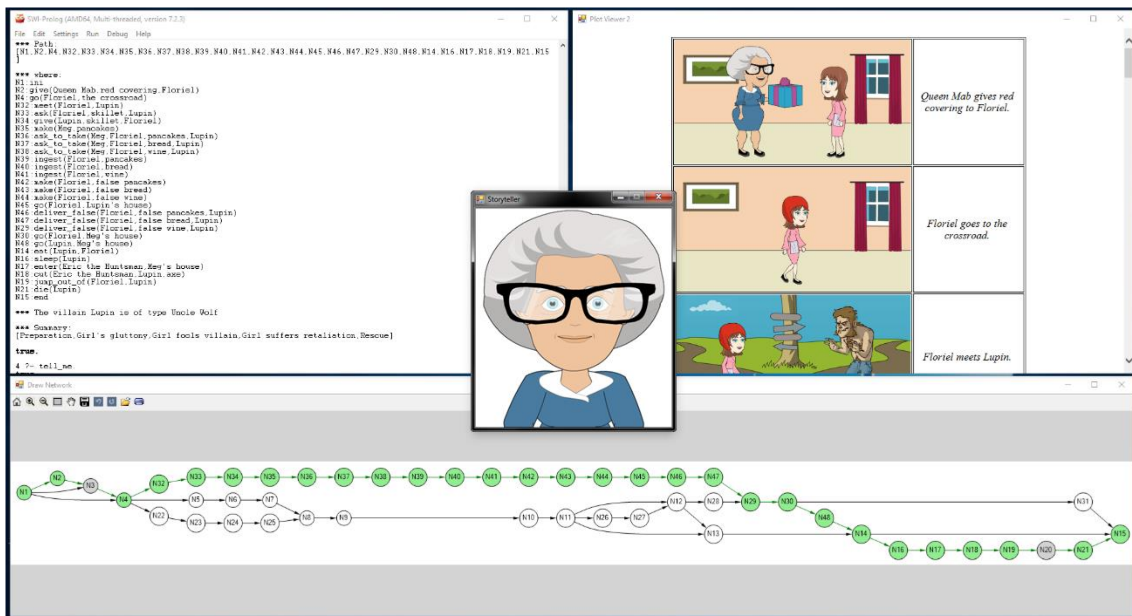

Figure 6 – user interface

The next sub-sections describe the modules in more detail.

5.1 Network Structure Visualization - Draw Network module

The process to visually display the network on the screen is implemented in the Draw Network module, which is written in C# and integrated with the SWI-Prolog environment through command-line arguments. In this module, the network is modeled as a directed graph $G = (N, E)$, where N is a set of event nodes, and E is a set of node pairs representing the event edges. In order to initialize the module, three arguments are required: (1) a node description list; (2) an edge list; and (3) an optional list of coloured nodes.

The node description list has the format:

$$[N_1:N_1^d,\ N_2:N_2^d,\ \ldots,\ N_n:N_n^d]$$

where N_i represents the name of the node and N_i^d is the description of the event(s) associated with the node. The names are used to identify the nodes and the descriptions are used to

create tooltips to be displayed to the user when, to learn about a given node, the user places the mouse cursor over the node's representation in the network drawing.

The second argument establishes the structure of the network through an edge list in the format:

$$[[N_1, N_2], [N_1, N_3], ..., [N_n, N_m]]$$

where N_i represents the name of the node. In the expression above, node N_1 is adjacent to nodes N_2 and N_3, and node N_n is adjacent to node N_m.

The third argument of the Draw Network module comprises a list of nodes and colours in the format:

$$[N_1{:}C_1, N_2{:}C_1, N_3{:}C_2, ..., N_n{:}N_m]$$

where N_i is the node name and C_i is the colour of the node, which is initially white (no colour) and changes to reflect plot-generation. As depicted in Figure 6, the nodes and edges in the path that is traversed to create a plot are coloured in light green, except nodes N3 and N2, which are coloured gray to mark their exclusion from the plot, as explained in section 4. Another colour, yellow, is used to point out the alternative nodes that can be chosen at a decision stage.

The process to visually draw the network and automatically adjust its layout to the screen is implemented using the Microsoft Automatic Graph Layout Library[2].

During plot-generation, the network is updated after each user decision stage through the same command-line arguments used to initialize the module, which creates another process of the Draw Network module. This new process automatically detects when another instance of its own process is already running and, instead of creating a new window to display the updated network, sends a message with the new network structure to the current Draw Network process, which updates and displays the new network. This inter-process communication is performed through a shared memory area [23].

5.2 Comics Dramatization – Plot Viewer module

The process to represent the generated stories in a comics-based story-board format is implemented in the Plot Viewer module. This module, written in C#, offers a storyboard-like comic strip representation for the generated stories, where each story event gains a graphical illustration and a short sentence description. Just like the Draw Network module, the Plot Viewer is equally integrated with the SWI-Prolog environment through command-line arguments.

To initialize the module and create a new story-board, three arguments are required: (1) a list of story events; (2) a list of characters and roles, which maps specific roles into customizable character names; and (3) an optional list of character variations, which allow the same character to have different appearances in different stories.

The list of story events is a time-ordered set of terms in the format:

$$\begin{aligned}[&ev_1(ev_1^{p1}, ev_1^{p2}, ..., ev_1^{pn}),\\ &ev_2(ev_2^{p1}, ev_2^{p2}, ..., ev_2^{pn}),\\ &...\\ &ev_m(ev_m^{p1}, ev_m^{p2}, ..., ev_m^{pn})]\end{aligned}$$

[2] Microsoft Automatic Graph Layout Library - https://github.com/Microsoft/automatic-graph-layout

where ev_i denotes the type of the event and the values of the ev_i^{pj} parameters define the characters, objects and places involved in the event ev_i. For each story event included in this list, a graphical illustration and a textual description of the event will be generated and added to the story-board.

The second parameter comprises a list that maps character's roles defined in the story domain into specific character names assigned by the user. This list has the following format:

$$[R_1:Na_1,\ R_2:Na_2,\ \ldots,\ R_n:Na_n]$$

where R_i is the role name and Na_i is the character name associated with role R_i. This information helps the Plot Viewer to identify proper graphical resources for the character names assigned by the user during the plot-generation phase.

The last argument to initialize the Plot Viewer is a list of character variations used to define distinct characterizations for specific characters of the story. This list has the following format:

$$[Na_1:C_1,\ Na_2:C_2,\ \ldots,\ Na_n:C_n]$$

where Na_i is the character name and C_i is the current characterization of N_i. With this information, the same character can be represented by different graphical resources in different stories. For example, the role of villain in the **LRRH** variants can be played by a Wolf, a Bzou (werewolf), or by an unforgiving Uncle Wolf (strange old man), which have different visual appearances. By using this information, the Plot Viewer can identify the appropriate graphical resources to represent the villain in each **LRRH** variant.

The story-board is created based on a context that defines how each class of events is visually represented. This context is specified in a XML file, which is composed of five main elements:

- **Options:** defines the style (bold/italic) and the size of the font used to describe the story events textually;

- **Accessories:** defines a set of visual elements (props, in theatrical parlance) that can be used by the story characters in certain events. For example, in the **LRRH** context, the girl sometimes carries a gift to her grandmother. This gift must be present in all illustrations of the girl in events between the moment when she is asked to take the gift and the moment when she delivers the gift. Each accessory has a `start` and an `end` event, indicating, respectively, the class of event when a character acquires the accessory and the class of event when the accessory is released. In addition, each accessory has a set of operator variations, which specify the position (`x` and `y`) where the image of the accessory must be drawn in the illustrations that include the character holding the accessory (usually this position fits the location of the character's hand);

- **Locations:** defines the visual aspects of the places where the story events can occur by associating each location with a specific background image;

- **Initial State:** defines the initial state for characters and objects that are susceptible to changes that affect the way the story events are visually represented. For example, the current location of each character is an essential information to establish the correct background image used to represent the place where the story events are deployed. Each state is specified by a term in the form `st(`p_1`, `p_2`, ..., `p_n`)`, where each `st` denotes the class of the state, and the values of the p_i parameters serve to characterize the instance of the state. In addition, each state has a `modifier`, which indicates the classes of events that can affect and modify the state;

- **Operators:** establishes how each class of events is visually and textually composed, which includes the definition of positions (`x` and `y`) where the visual elements (images of characters and objects) are placed on the illustrations, and the template (`TextTemplate`) used to generate a textual description of the event. The operators are defined by a term in the form `ev(`p_1`, `p_2`, …, `p_n`)`, and are associated with a set of visual elements (image files) whose file names are dynamically composed according to a template based on the p_i parameters of the operator (written in the format `%`p_i`%`). For example, the operator `meet`, which represents the event where a character meets another character, is defined by the term `meet(`p_1`, `p_2`)` and is associated with two graphical resources: the image of the character p_1 facing right (`image_%`p_1`%_normal_right.png`) and the image of the character p_2 facing left (`image_%`p_2`%_normal_left.png`). When the operator `meet` is instantiated as a story event, the `%`p_i`%` parameters in the resources' templates are replaced by their respective p_i values. For the event `meet(girl, wolf)`, the resulting file names are: `image_girl_normal_right.png` and `image_wolf_normal_left.png`.

The structure of the XML file that describes the context is illustrated in Figure 7. An example of context file used in the LRRH domain is shown in Appendix 2.

To generate graphical illustrations for the story events, the Plot Viewer implements a simple scene compositing algorithm that combines multiple images of characters, objects and locations in a single image (panel). The process to compose the illustration of a story event comprises seven steps:

1. Create the an empty image for the panel;
2. Identify the location where the story event is occurring based on the current state of the characters involved in the event;
3. Draw the background illustration of the location over the panel image;
4. Identify the generic operator that describes the visual elements necessary to represent the story event;
5. Draw all the visual elements over the panel according to the parameters of the story event and the positions defined in the generic operator;
6. Identify the accessories held by characters involved in the story event;
7. Draw all the accessories over the panel according to the positions defined in the story context for the current story event.

The textual descriptions for the story events are created through a simple template-based method that uses the text templates defined in the story context to create short text sentences describing the story events. For example, if we have a template for the operator `meet` defined as "p_1` meets `p_2`.`" and the event `meet(Little Girl, Bzou)`, the sentence

"Little Girl meets Bzou." will be generated by replacing the p_i parameters by their respective values in the event instance.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<PlotViewer context="name">
   <Options>
     <FontSize>number</FontSize>
     <FontBold>boolean</FontBold>
     <FontItalic>boolean</FontItalic>
   </Options>
   <Accessories>
     <Accessory start="evi" end="evj" file="accessory_image.png">
        <Operator name="evi" x="number" y="number"/>
        .
          .
          .
     </Accessory>
   </Accessories>
   <Locations>
     <Location name="loc1" file="location_image.png"/>
       .
       .
       .
    </Locations>
   <InitialState>
     <StateItem value="sti(p1, p2, …, pn)" modifier="ev1, ev2, …,evn"/>
       .
       .
       .
   </InitialState>
   <Operators>
     <Operator name="evi(p1, p2, …, pn)">
        <Element file="elem_%p1%_image.png" x="number" y="number"/>
          .
          .
          .
        <TextTemplate template="%p1% do %p2% to %pn%."/>
     </Operator>
   </Operators>
</PlotViewer>
```

Figure 7 – structure of the XML file that describes a story context

The story-board is created and displayed as a HTML page, where the panel images generated by the scene compositing algorithm are linked with their respective textual descriptions. The HTML code for the story-board is automatically generated by the Plot Viewer according to the story events.

More details on the generation of comic strips can be found in our previous work on interactive comics [25].

5.3 From templates to meta-templates

Our prototype includes, within its SWI Prolog Interface, a module for generating textual representations of plots that more closely resemble natural language. The module makes

use of *meta-templates* for the compound production of story text, as opposed to sequences of single sentences.

Meta-templates are templates for combining basic templates, like the ones used by the Plot Viewer module. For example, the event `make('Floriel', 'false pancakes')` generates through a *basic template* the sentence "`Floriel prepares false pancakes.`". Thus, the sequence of events:

```
[make('Floriel','false pancakes'),
 make('Floriel','false bread'),
 make('Floriel','false wine')]
```

without any sort of post-processing, yields "`Floriel prepares false pancakes. Floriel prepares false bread. Floriel prepares false wine`".

This piecewise representation is suitable for the comics dramatization, where the art of each panel varies and there is meaning in representing these events in separate sentences. However, when reading an entire story continuously, the desired representation would be one that "weaves" these operations into a compact form: "`Floriel prepares false wine, false pancakes and false bread.`".

This particular result is achieved, in our module, through the application of meta-templates which recognize that the same operation (make) is performed by the same subject (Floriel) with varying objects (false pancakes, false bread and false wine), resulting in a single operation where the object is a compound term. We have defined meta-templates for dealing with cases such as the above; and also for cases where multiple agents perform the same operation, and for cases where multiple agents perform operations over multiple objects.

We have also defined meta-templates for generating pronoun references, which are applied in conjunction and combination with the meta-templates above. Thus, one possible resulting story text for:

```
[go('Little Girl', 'Grandmother\'s house'),
 deliver('Little Girl',food)]
```

is "Little Girl goes to Grandmother's house. She delivers food.". This avoids the repetitive use of entity names when narrating the story, which becomes especially important for our audio narration module.

Our meta-template module is also capable of generating textual sequences of facts within a domain, but that is outside of the scope of the present work, where we focus on events and sequences of events. Just as happens in the network generation module, all meta-template clauses that are specific to the domain are stored separately, so that the generic module can be applied to any domain. Examples of all cases described above can be seen in Appendix 3, where meta-templates are applied to events in an academic domain.

The story text generated by the meta-template module is provided to the Storyteller module, described below.

5.4 Audio Narration – Storyteller module

The process to tell the generated stories vocally is implemented in the Storyteller module, which is written in C# and integrated with the SWI-Prolog environment through a command-line argument with the story described in a text format.

In order to convert the story into speech, the Storyteller module uses the text-to-speech (TTS) engine that comes integrated with the Windows operational system[3]. The TTS engine recognizes the story text and, using the standard synthesized female voice, narrates the story. In addition, a simple animation of the grandmother with lip-sync movements is shown while the story is being narrated in response to a `tell_me` user command.

The same engine is applied for the oral narration of the stories kept in a personal library, composed with the help of meta-templates as explained in the previous sub-section.

6. Related work

As remarked in the Introduction, our work is inspired on and related to the literary studies and classification systems [34,7,2,43] mentioned throughout the text. As we deal with the analysis and composition of stories through computational means, we also position this paper in relation to works in the areas of Interactive Storytelling and Computational Narratology.

While Computational Narratology refers to the broad study of narratives interpreted, generated and explored within different genres [29,12], Interactive Storytelling relies on the underlying narrative theories to implement digital forms of entertainment [13,38], as well as training, education [24] and *serious games* [20]. Our network approach can be extended to serious games, in that it can be applied over sequences of events extracted from database logs, ready to be processed by Process Mining techniques [1]. The generation of new stories – or *traces* not present in the database itself – relates to simulation [39,15].

Many works in these domains adopt the representation of narrative plots as sequences of events. In particular, there are numerous *plot-based* approaches [36] where automated planning techniques are used for the (semi-)automatic composition of plots with variability and coherence [14,30,3].In these works, stories typically emerge from the combinations of *events* as atomic elements, and not from blending [18] pre-existing variants.

Another aspect of our work that is related to Interactive Storytelling systems is that of plot dramatization, under the form of text, oral narration and comics [39,44,25,4]. The automatic generation of comics has been an active topic of research since Kurlander et al. proposed their famous Comic Chat system [22] in the nineties. Interactive Storytelling systems explore this form of dramatization in different ways, such as presented by Alves et al. [4], where summaries of agent-based stories generated by the system FearNot! [5] are presented as comic strips.

One of the core concepts in the present work is the representation of narratives as networks – graphs establishing ordering relations between events – for plot analysis and composition. This is related to non-linear approaches in Interactive Storytelling systems [9]. Those approaches differ from ours in that they overload the graph with information other than at the *fabula* level, including information about user interaction mediation [37], non-determinism at the execution level and dramatization information [40], or else focus only on the generation of alternative plots [31]. One approach that, similarly to ours, uses graph representation of events for plot analysis, is that of Purdy and Riedl [35], which is based, however, on drawing inferences from perceived causality between events in large corpora of stories, and not on the generalization of variants within a genre.

[3] https://support.microsoft.com/en-us/help/306902/how-to-configure-and-use-text-to-speech-in-windows-xp-and-in-windows-vista

7. Concluding remarks

Plot composition by reusing previous variants, as we propose to do via our network-based method, is not really a novelty. Even professional authors do not hesitate to reuse any number of attractive little pieces extracted from other peoples' narratives, which seem to be made to fit together, to help composing their own texts, as Roland Barthes has famously expressed [8, p. 39]:

> Any text is a new tissue of past citations. Bits of code, formulae, rhythmic models, fragments of social languages, etc., pass into the text and are redistributed within it, for there is always language before and around the text.

This is particularly true in the case of folktale genres. Moved by a compulsion that Alfred Lord called *tension of essences* [28], the popular oral storytellers responsible for their dissemination did faithfully preserve their traditional formulae – types and motifs – thus guaranteeing that any new variant remains within the scope of the genre. Since those storytellers usually dominated an ample repertoire of tales of different types, it should be common practice among them to combine fragments from different tales to charm even more their audience. For example, it has been noted [41] that Perrault's variant may represent a more ancient – hence of more "authentic" popular origin – rendering of **LRRH** than that collected by the Grimm brothers, which may have been the result of blending the former with *The Wolf and the Seven Little Kids* (**type 123**), also present in the Grimm collection [21, p. 39]. Modern storytelling does also endorse this mutual *type contamination* trend, as demonstrates the highly successful *Into the Woods*[4] musical drama, whose cast was recruited not only from *Little red Riding Hood*, but also from such disparate folktale sources as *Jack and the Beanstalk*, *Rapunzel*, and *Cinderella*.

So, for non-professional authors, such as the casual users that we had in mind in our project, there is even more reason to offer interactive systems that, while encouraging them to compose plots that suit their taste, make recommendations and impose restrictions in order to preserve the conventions of the (sub-)genre on hand. We believe that the work described in the present paper is a useful contribution in this direction.

Having formulated and implemented the general algorithms needed to construct networks from variants and generate new plots, we prepared the **LRRH** application module without much difficulty. More research is needed, however, to develop authorial tools – that do not require knowledge of programs and formalisms – to effectively help domain experts to configure specialized modules appropriate to handle their domains.

References

1. Aalst van der, W. M. Extracting event data from databases to unleash process mining. In *BPM-Driving innovation in a digital world*. Springer, 2015.
2. Aarne, A and Thompson, S. *The Types of the Folktale*. Academia Scientiarum Fennica, 1961.
3. Abelha, P., et al. A nondeterministic temporal planning model for generating narratives with continuous change in interactive storytelling. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[4] https://www.ibdb.com/broadway-show/into-the-woods-4753

4. Alves T., Simões A., Figueiredo R., Vala M., Paiva A., Aylett R. So tell me what happened: Turning agent-based interactive drama into comics. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, 2008.
5. Aylett, R., Louchart, S., Dias, J., Paiva, A. FearNot! An Experiment in Emergent Narrative. In *Proceedings of the 5th International Conference on Intelligent Virtual Agents*, 2005.
6. Bach, B. et al. Telling stories about dynamic networks with graph comics. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016.
7. Bal, M. *Narratology*. University of Toronto Press, 1997.
8. Barthes, R. Theory of the Text. In *Untying the Text: a Post-Structuralist Reader*. Young, J.C. (ed.). Routledge & Kegan Paul, 1981.
9. Bosser, A. G., Cavazza, M. O., Champagnat, R. Linear logic for non-linear storytelling. *IOS Press*, 2010.
10. Calvino, I. *Italian Folktales*. Mariner Books, 1992.
11. Casati, F., Ceri, S., Pernici, B., Pozzi, G. Conceptual modeling of workflows. In *Proceedings of the International Conference on Conceptual Modeling (ER)*, 1995.
12. Cavazza, M. Pizzi, D. Narratology for interactive storytelling: a critical introduction. In *Proceedings of the Third International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, 2006.
13. Camanho, M.M., Ciarlini, A.E.M., Furtado, A.L., A model for Interactive TV Storytelling. In: *Proceedings of the 7th Brazilian Symposium on Games and Digital Entertainment. Rio de Janeiro*, 2009.
14. Charles, F., Lozano, M., Mead, S. J., Bisquerra, A. F., Cavazza, M. Planning formalisms and authoring in interactive storytelling. In *Proceedings of 3rd Technologies for Interactive Digital Storytelling and Entertainment*, 2006.
15. Ciarlini, A.E.M., Casanova, M.A., Furtado, A.L., Veloso, P.A.S. Modeling interactive storytelling genres as application domains. *Journal of Intelligent Information Systems*, 35(3), 2010.
16. Ciarlini, A.E.M., Pozzer, C.T., Furtado, A.L., Feijo, B. A logic-based tool for interactive generation and dramatization of stories. In *Proceedings of the ACM-SIGCHI International Conference on Advances in Computer Entertainment Technology*, 2005.
17. Delarue, P. The Story of Grandmother. In *Little Red Riding Hood: A Casebook*. Dundes, A. (ed). University of Wisconsin Press, 1989.
18. Fauconnier, G. Turner, M. *Conceptual projection and middle spaces*. Tech. Rep. 9401, Univ. California, San Diego, 1994.
19. Fikes, R. E. Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* , 2(3-4), 1971.
20. Gottin, V. M, Lima, E. S., Furtado, A. L. Applying Digital Storytelling to Information System Domains. *Proceedings of the XIV Brazilian Symposium on Computer Games and Digital Entertainment*, p. 192-195, 2015.
21. Grimm, J. Grimm, W. *The Complete Grimm's FairyTales*. M. Hunt and J. Stern (trans.). Pantheon, 1972.
22. Kurlander, D., Skelly, T., Salesin, D. Comic Chat. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996.
23. Lamport, L. *On interprocess communication*. Distributed Computing, 1(2), p. 77- 85, 1986.

24. Laurent, M., Szilas, N., Lourdeaux, D., Bouchardon, S. A Reflexive Approach in Learning Through Uchronia. In *Proceedings of the 9th International Conference on Interactive Digital Storytelling*, Springer, 2016.
25. Lima, E.S., Feijó, B., Furtado, A.L., Barbosa, S.D.J., Pozzer, C.T., Ciarlini, A. Non-Branching Interactive Comics. In *Proceedings of the 10th International Conference on Advances in Computer Entertainment Technology*, pp. 230-245, 2013.
26. Lima, E.S., Feijó, B., Casanova, M.A., Furtado, A.L. Storytelling Variants Based on Semiotic Relations. *Entertainment Computing*, 17, p. 31-44, 2016.
27. Lima, E.S., Furtado, A.L., Feijó, B. Storytelling variants: the case of Little Red Riding Hood. In *Proceedings of the 14th International Conference on Entertainment Computing*, pp. 286-300, 2015.
28. Lord, A. *The Singer of Tales*. Harvard University Press, 2000.
29. Mani, I. *Computational Narratology*. In: Hühn, Peter et al. (eds.): the living handbook of narratology. Hamburg: Hamburg University Press, 2014.
30. Mateas, M., Stern, A. Structuring Content in the Façade Interactive Drama Architecture. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment,* 2005.
31. Min, W. H., Shim, E. S., Kim, Y. J., Cheong, Y. G. Planning-integrated story graph for interactive narratives. In *Proceedings of the 2nd ACM international workshop on Story representation, mechanism and context*, ACM, 2008.
32. Orenstein, C. *Little Red Riding Hood Uncloaked*. Basic Books, 2003.
33. Perrault, C. Little Red Riding Hood. In *Beauties, Beasts and Enchantment - Classic French Fairy Tales*. J. R. Planché and J. Zipes (trans.). Meridian, 1991.
34. Propp, V. *Morphology of the Folktale*. Laurence, S. (trans.), University of Texas Press, 1968.
35. Purdy, C., Riedl, M. Reading Between the Lines: Using Plot Graphs to Draw Inferences from Stories. In *Proceedings of the 9th International Conference on Interactive Digital Storytelling*, 2016.
36. Riedl, M. Stern, A., Believable agents and intelligent story adaptation for Interactive Storytelling. In *Proceedings of the Third International Conference on Technologias for Interactive Digital Storytelling and Entertainment*. Springer, 2006.
37. Riedl, M., Young, R. From linear story generation to branching story graphs. *IEEE Computer Graphics and Applications*, 26(3), 2006.
38. Samuel, B., et al. Bad News: An experiment in computationally assisted performance. In *Proceedings of the 9th International Conference on Interactive Digital Storytelling*, 2016.
39. Swartjes, I. M. T., Theune, M. The virtual storyteller: Story generation by simulation. In *Proceedings of the 20th Belgian-Netherlands Conference on Artificial Intelligence*, 2008.
40. Swartjes, I., & Theune, M. A fabula model for emergent narrative. In *International Conference on Technologies for Interactive Digital Storytelling and Entertainment*. Springer Berlin Heidelberg, 2006.
41. Tehrani, J.J. The Phylogeny of Little Red Riding Hood. *PLOS ONE*, v. 8, 11, 2013.
42. Thompson S. *The Folktale*. University of California Press, 1977.
43. Uther, H.J. *The Types of International Folktales*. Finish Academy of Science and Letters, 2011.
44. Zeeders, R.. *Comics-comic generation from story content graphs*. Master's thesis, University of Twente, Department of Electrical Engineering, Mathematics and Computer Science, 2010.

Appendix 1 – The LRRH variants utilized

Perrault -

```
[give('Grandmother', 'little red hood', 'Little Red Riding Hood'),
ask_to_take('Mother', 'Little Red Riding Hood', 'cake and butter',
'Grandmother'),  go('Little Red Riding Hood', 'the woods'), meet('Little
Red Riding Hood', 'Wolf'), go('Wolf', 'Grandmother\'s house'),
eat('Wolf', 'Grandmother'), disguise('Wolf', 'Grandmother'),
lay_down('Wolf', 'Grandmother\'s bed'), go('Little Red Riding Hood',
'Grandmother\'s house'), deliver('Little Red Riding Hood', 'cake and
butter'), lay_down('Little Red Riding Hood', 'Grandmother\'s bed'),
question('Little Red Riding Hood', 'Wolf'), eat('Wolf', 'Little Red
Riding Hood')].
```

Grimm brothers -

```
[give('Grandmother', 'red velvet cap', 'Little Red Cap'),
ask_to_take('Mother', 'Little Red Cap', 'cake and wine', 'Grandmother'),
go('Little Red Cap', 'the woods'), meet('Little Red Cap', 'Wolf'),
go('Wolf', 'Grandmother\'s house'), eat('Wolf', 'Grandmother'),
disguise('Wolf', 'Grandmother'), lay_down('Wolf', 'Grandmother\'s bed'),
go('Little Red Cap', 'Grandmother\'s house'), deliver('Little Red Cap',
'cake and wine'), question('Little Red Cap', 'Wolf'), eat('Wolf', 'Little
Red Cap'), sleep('Wolf'), go('Hunter', 'Grandmother\'s house'),
cut('Hunter', 'Wolf', axe), jump_out_of('Little Red Cap', 'Wolf'),
jump_out_of('Grandmother', 'Wolf'), die('Wolf')].
```

Paul Delarue -

```
[ask_to_take('Mother', 'Little Girl', 'loaf and milk', 'Grandmother'),
go('Little Girl', 'the crossroad'), meet('Little Girl', 'Bzou'),
pick('Little Girl', needles, forest), go('Bzou', 'Grandmother\'s house'),
kill('Bzou', 'Grandmother'), disguise('Bzou', 'Grandmother'),
 lay_down('Bzou', 'Grandmother\'s bed'), go('Little Girl',
'Grandmother\'s house'), deliver('Little Girl', 'loaf and milk'),
ingest('Little Girl', 'Grandmother\'s flesh'), undress('Little Girl'),
lay_down('Little Girl', 'Grandmother\'s bed'), question('Little Girl',
'Bzou'), fool('Little Girl', 'Bzou'), go('Little Girl', 'Mother\'s
house'), go('Bzou', 'Mother\'s house')].
```

Italo Calvino -

```
[go('Little Girl','Uncle Wolf\'s house'),meet('Little Girl', 'Uncle
Wolf'), ask('Little Girl', skillet, 'Uncle Wolf'),give('Uncle Wolf',
skillet,'Little Girl'),  make('Mother', pancakes), ask_to_take('Mother',
'Little Girl', pancakes, 'Uncle Wolf'),ask_to_take('Mother', 'Little
Girl', bread, 'Uncle Wolf'), ask_to_take('Mother', 'Little Girl',
wine,'Uncle Wolf'),
 ingest('Little Girl', pancakes), ingest('Little Girl', bread),
ingest('Little Girl',wine), make('Little Girl', 'false pancakes'),
make('Little Girl', 'false bread'), make('Little Girl', 'false wine'),
go('Little Girl','Uncle Wolf\'s house'),deliver('Little Girl', 'false
pancakes','Uncle Wolf'),deliver('Little Girl', 'false bread', 'Uncle
Wolf'), deliver('Little Girl', 'false wine','Uncle Wolf'), go('Little
Girl', 'Mother\'s house'), go('Uncle Wolf', 'Mother\'s house'),
eat('Uncle Wolf', 'Little Girl')].
```

Appendix 2 – LRRH context file


```
<?xml version='1.0' encoding='ISO-8859-1'?>
<PlotViewer context="Little Red Riding Hood">
  <Options>
    <FontSize>18</FontSize>
    <FontBold>false</FontBold>
    <FontItalic>true</FontItalic>
  </Options>
  <Accessories>
    <Accessory start="ask_to_take" end="deliver"
                                  file="Data\\food_accessory.png">
      <Operator name="go" x="125" y="60"/>
      <Operator name="meet" x="20" y="60"/>
    </Accessory>
  </Accessories>
  <Locations>
    <Location name="Mother's house" file="Data\\Mother_BG.png"/>
    <Location name="Grandmother's house" file="Data\\Grand_BG.png"/>
    <Location name="the woods" file="Data\\Woods_BG.png"/>
    <Location name="the crossroad" file="Data\\Crossroad_BG.png"/>
    <Location name="villain's house" file="Data\\Villain_BG.png"/>
  </Locations>
  <InitialState>
    <State value="at(Mother, Mother's house)" modifier="go"/>
    <State value="at(Grandmother, Grandmother's house)"
                                                    modifier="go"/>
    <State value="at(victim, Mother's house)" modifier="go"/>
    <State value="at(villain, the woods)" modifier="go"/>
    <State value="at(Hunter, the woods)" modifier="enter"/>
  </InitialState>
  <Operators>
    <Operator name="give(A,B,C)">
      <Element file="Data\\%A%_give.png" x="30" y="0"/>
      <Element file="Data\\%B%.png" x="30" y="0"/>
      <Element file="Data\\%C%_receive.png" x="170" y="0"/>
      <TextTemplate template="%A% gives %B% to %C%."/>
    </Operator>
    <Operator name="ask_to_take(A,B,C,D)">
      <Element file="Data\\%A%.png" x="0" y="0"/>
      <Element file="Data\\%B%.png" x="200" y="0"/>
      <Element file="Data\\%C%.png" x="70" y="25"/>
      <TextTemplate template="%A% asks %B% to take %C% to %D%."/>
    </Operator>
    <Operator name="go(A,B)">
      <Element file="Data\\%A%_go.png" x="100" y="0"/>
      <TextTemplate template="%A% goes to %B%."/>
    </Operator>
    <Operator name="meet(A,B)">
      <Element file="Data\\%A%_meet.png" x="0" y="0"/>
      <Element file="Data\\%B%.png" x="200" y="0"/>
      <TextTemplate template="%A% meets %B%."/>
    </Operator>
    <Operator name="ask(A,B,C)">
      <Element file="Data\\%A%_go.png" x="0" y="0"/>
      <Element file="Data\\%C%.png" x="200" y="0"/>
      <TextTemplate template="%A% asks %C% for a %B%."/>
    </Operator>
    <Operator name="eat(A,B)">
```

```
    <Element file="Data\\%B%_scared.png" x="150" y="0"/>
    <Element file="Data\\%A%_eat.png" x="50" y="0"/>
    <TextTemplate template="%A% eats %B%."/>
  </Operator>
  <Operator name="kill(A,B)">
    <Element file="Data\\%B%_scared.png" x="140" y="0"/>
    <Element file="Data\\%A%_eat.png" x="50" y="0"/>
    <TextTemplate template="%A% kills %B%."/>
  </Operator>
  <Operator name="disguise(A,B)">
    <Element file="Data\\%A%_disguise.png" x="100" y="0"/>
    <TextTemplate template="%A% disguises as %B%."/>
  </Operator>
  <Operator name="lay_down(A,B)">
    <Element file="Data\\%A%_lay_down.png" x="200" y="40"/>
    <TextTemplate template="%A% lies down on %B%."/>
  </Operator>
  <Operator name="question(A,B)">
    <Element file="Data\\%A%_question.png" x="0" y="0"/>
    <Element file="Data\\%B%_question.png" x="200" y="40"/>
    <TextTemplate template="%A% questions %B%."/>
  </Operator>
  <Operator name="sleep(A)">
    <Element file="Data\\%A%_sleep.png" x="200" y="40"/>
    <TextTemplate template="%A% falls asleep."/>
  </Operator>
  <Operator name="make(A,B)">
    <Element file="Data\\%A%_make.png" x="85" y="0"/>
    <Element file="Data\\%B%.png" x="140" y="30"/>
    <TextTemplate template="%A% makes %B%."/>
  </Operator>
  <Operator name="deliver(A,B)">
    <Element file="Data\\%A%_deliver.png" x="80" y="0"/>
    <Element file="Data\\%B%.png" x="135" y="50"/>
    <TextTemplate template="%A% delivers %B%."/>
  </Operator>
  <Operator name="deliver_false(A,B,C)">
    <Element file="Data\\%A%_deliver.png" x="0" y="0"/>
    <Element file="Data\\%B%.png" x="47" y="50"/>
    <Element file="Data\\%C%.png" x="200" y="0"/>
    <TextTemplate template="%A% delivers %B% to %C%."/>
  </Operator>
  <Operator name="fool(A,B)">
    <Element file="Data\\%A%_fool.png" x="0" y="0"/>
    <Element file="Data\\%B%.png" x="200" y="0"/>
    <TextTemplate template="%A% fools %B%."/>
  </Operator>
  <Operator name="pick(A,B,C)">
    <Element file="Data\\%A%_pick.png" x="80" y="2"/>
    <Element file="Data\\%B%.png" x="120" y="40"/>
    <TextTemplate template="%A% pick %B%."/>
  </Operator>
  <Operator name="ingest(A,B)">
    <Element file="Data\\%A%_eat.png" x="100" y="0"/>
    <Element file="Data\\%B%.png" x="160" y="28"/>
    <TextTemplate template="%A% ingests %B%."/>
  </Operator>
  <Operator name="enter(A,B)">
    <Element file="Data\\%A%_go.png" x="100" y="0"/>
```

```
        <TextTemplate template="%A% enters %B%."/>
      </Operator>
      <Operator name="cut(A,B,C)">
        <Element file="Data\\%C%.png" x="45" y="-38"/>
        <Element file="Data\\%A%_cut.png" x="0" y="0"/>
        <Element file="Data\\%B%_cut.png" x="180" y="10"/>
        <TextTemplate template="%A% cuts %B% with an %C%."/>
      </Operator>
      <Operator name="jump_out_of(A,B)">
        <Element file="Data\\%A%_jump.png" x="50" y="0"/>
        <Element file="Data\\%B%_cut.png" x="170" y="10"/>
        <TextTemplate template="%A% jumps out of %B%."/>
      </Operator>
      <Operator name="die(A)">
        <Element file="Data\\%A%_cut.png" x="100" y="10"/>
        <TextTemplate template="%A% dies."/>
      </Operator>
      <Operator name="undress(A)">
        <Element file="Data\\%A%_undress.png" x="100" y="0"/>
        <TextTemplate template="%A% undresses."/>
      </Operator>
    </Operators>
</PlotViewer>
```

Appendix 3 – Meta-templates examples

```
Operations that are unchanged
 [hire(John,assistant)]
     John was hired, as assistant-professor.

 [mark(Adam,Logic,pass)]
      Adam's mark in Logic was 'pass'.

Same operation, same agent, multiple objects
[enroll(Bea,Art),enroll(Bea,Design),enroll(Bea,Semiotics)]
     Student Bea enrolled in Semiotics, Art and Design.

[drop(Bea,Art),drop(Bea,Music),drop(Bea,Semiotics)]
     Student Bea dropped Semiotics, Art and Music.

[transfer(Bea,Art,Logic),transfer(Bea,Music,Math),transfer(Bea,Semiotics,
Physics)]
     Student Bea transferred from Semiotics, Art and Music  to Physics,
     Logic and Math.

[pass(Bea,Logic,0,2),pass(Adam,Logic,0,2)]
     Bea and Adam, having passed course Logic, have a total of 2 credits.

[create_program(Alpha,5),create_program(Beta,5)]
     Alpha and Beta are  open, requiring a total of  5 credits.

[receive_degree(Bea,Alpha),receive_degree(Bea,Beta)]
     Student  Bea has graduated in Alpha and Beta.

Same operation, multiple agent, same objects
[enroll(Bea,Art),enroll(Adam,Art),enroll(John,Art)]
     John, Bea and Adam enrolled in course Art .

[drop(Bea,Art),drop(Adam,Art),drop(John,Art)]
     John, Bea and Adam dropped course Art .

[transfer(Bea,Art,Logic),transfer(Adam,Art,Logic),transfer(John,Art,Logic
)]
     John, Bea and Adam transferred from course Art   to course Logic.

[cancel(Art),cancel(Music)]
     Art and Music are cancelled.

[change_cr(Art,1,2),change_cr(Music,1,2)]
     Number of credits of Art and Music changed from  1  credits, to  2
     credits.

Same operation, multiple agents and multiple objects
[enroll(Bea,Art),enroll(Bea,Design),enroll(Adam,Art),enroll(Adam,Design)]
     Bea and Adam enrolled in Art and Design.

[enroll(Bea,Art),enroll(Bea,Design),enroll(Adam,Design),enroll(Adam,Art)]
     Bea and Adam enrolled in Art and Design.

[enroll(Bea,Art),enroll(Adam,Art),enroll(Bea,Design),enroll(Adam,Design)]
     Bea and Adam enrolled in Art and Design.

[enroll(Bea,Art),enroll(Adam,Design),enroll(Bea,Design),enroll(Adam,Art)]
```

```
Student Bea enrolled in course Art . Adam and Bea enrolled in course
Design . Student Adam enrolled in course Art .
```

```
[pass(Bea,Logic,0,2),pass(Adam,Logic,1,3)]
    Student  Bea, having passed course Logic, has a total of 2 credits.
    Student  Adam, having passed course Logic, has a total of 3 credits.
```

Pronoun references

```
[enroll(Bea,Art),drop(Bea,Design)]
    Student Bea enrolled in course Art . She dropped course Design .

    Student Bea enrolled in course Art . Student Bea dropped course
    Design .

[enroll(Bea,Art),drop(Bea,Design),enroll(Bea,Logic)]
    Student Bea enrolled in course Art . She dropped course Design . and
    enrolled in course Logic .

    Student Bea enrolled in course Art . She dropped course Design . She
    enrolled in course Logic .

    Student Bea enrolled in course Art . She dropped course Design .
    Student Bea enrolled in course Logic .

    Student Bea enrolled in course Art . Student Bea dropped course
    Design . She enrolled in course Logic .

    Student Bea enrolled in course Art . Student Bea dropped course
    Design . Student Bea enrolled in course Logic .

[enroll(Bea,Art),drop(Bea,Design),transfer(Bea,Logic,Semiotics)]
    Student Bea enrolled in course Art . She dropped course Design . and
    transferred from course Logic   to course Semiotics .

    Student Bea enrolled in course Art . She dropped course Design . She
    transferred from course Logic   to course Semiotics .

    Student Bea enrolled in course Art . She dropped course Design .
    Student Bea transferred from course Logic   to course Semiotics .

    Student Bea enrolled in course Art . Student Bea dropped course
    Design . She transferred from course Logic   to course Semiotics .

    Student Bea enrolled in course Art . Student Bea dropped course
    Design . Student Bea transferred from course Logic   to course
    Semiotics .

[cancel(Art),offer(Art,2,Q,Z)]
    Course  Art is cancelled. Course  Art is created with 2 credits,
    being taught by Q who uses Z's book.

[transfer(Bea,Art,Logic),transfer(Adam,Art,Logic),transfer(John,Art,Logic
),drop(Bea,Music),drop(Adam,Music),drop(John,Music)]
    John, Bea and Adam transferred from course Art   to course Logic .
    They dropped course Music.

    John, Bea and Adam transferred from course Art   to course Logic .
    John, Bea and Adam dropped course Music.
```

[transfer(Bea,Art,Logic),transfer(Adam,Art,Logic),transfer(John,Art,Logic),drop(John,Music),drop(Adam,Music),drop(Bea,Music)]

John, Bea and Adam transferred from course Art to course Logic . They dropped course Music .

John, Bea and Adam transferred from course Art to course Logic . Bea, John and Adam dropped course Music .

[enroll(Bea,Art),enroll(Bea,Design),enroll(Bea,Semiotics),drop(Bea,Art),drop(Bea,Music),drop(Bea,Semiotics)]

Student Bea enrolled in Semiotics, Art and Design. She dropped Semiotics, Art and Music.

Student Bea enrolled in Semiotics, Art and Design. Student Bea dropped Semiotics, Art and Music.