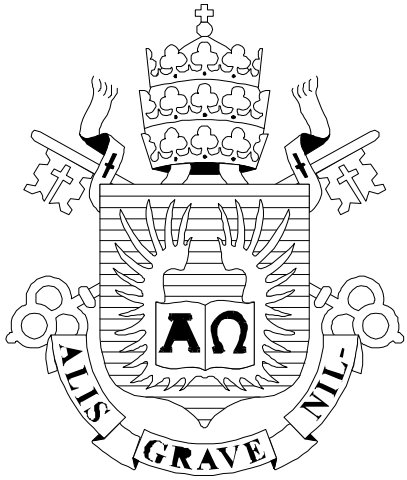

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 01/09

PMA – A Plot-Manipulation Algebra

Börje Felipe Fernandes Karlsson **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. 01/09 ISSN 0103-9741
Editor: Prof. Carlos José Pereira de Lucena January 2009

PMA – A Plot-Manipulation Algebra

Börje Felipe Fernandes Karlsson Antonio L. Furtado

{borje,furtado}@inf.puc-rio.br

Abstract: Plot composition is examined here at a logic design level, an intermediate stage that comes next to the conceptual level wherein the intended narrative genre is specified. An abstract data structure is proposed to represent plots, together with an algebra for manipulating the data structure. Our purpose is to adapt for narratives the strategy applied to databases by Codd's relational model. The basic operators of our Plot-Manipulation Algebra (PMA) were introduced in view of the four fundamental relations between events that we identified in a previous work. A logic programming prototype was implemented, in order to run examples using the algebra.

Keywords: storytelling, narratology, plots, logic design, algebraic formalisms, logic programming.

Resumo: A composição de enredos é aqui examinada ao nivel de projeto lógico, estágio intermediário que se segue ao nivel conceitual em que o gênero pretendido de narrativas é especificado. Uma estrutura abstrata de dados é proposta para representar enredos, juntamente com uma álgebra para manipular a estrutura de dados. Nosso propósito é adpatar para narrativas a estratégia aplicada a bancos de dados pelo modelo relacional de Codd. Os operadores básicos de nossa Álgebra de Manipulação de Enredos (PMA) foram introduzidos em vista das quatro relações fundamentais entre eventos que identificamos em trabalho anterior. Um protótipo em linguagem de programação em lógica foi implementado, para rodar exemplos utilizando a álgebra.

Palavras-chave: narração de estórias, narratologia, enredos, projeto lógico, formalismos algébricos, 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

Narratology studies distinguish three levels in literary composition: fabula, story and text [Bal]. In this paper, we stay at the *fabula* level, where the characters acting in the narrative are introduced, as well as the narrative *plot*, consisting of a partially-ordered set of *events*.

At least four concerns are involved in plot composition:

- the plot must be formed by a *coherent sequence* of events;
- for each position in the sequence, several *alternative choice*s may be applicable;
- non-trivial interesting sequences must permit unexpected *shifts along the way*;
- one may need to *go down to details* to better visualize the events.

These concerns led us to identify, drawing on linguistic and semiotic research [Saussure; Booth; Chandler], four relations between events, called, respectively, *syntagmatic*, *paradigmatic*, *antithetic*, and *meronymic* relations [Ciarlini & al, 2008]. It turns out that such relations hold as a consequence of the conventions regulating the chosen narrative *genre*. Therefore a necessary preliminary step to plot composition is to provide a *conceptual specification* of the intended genre.

In our conceptual modelling approach, we focus on events that correspond to the execution of predefined *operations*, deliberately performed by the characters. Each operation is defined in terms of its pre-conditions and post-conditions, and the interplay of the pre-/ post-conditions is what induces the partial order requirements for the plots, and, furthermore, constitutes the basis for characterizing the presence of the four kinds of relations between events.

However a conceptual specification is still too far removed from a concrete computerized system to support plot composition. An analogous problem was successfully faced by database researchers, and, as we shall indicate, their three-stage solution can be conveniently adapted for our purposes. The key idea is to provide a *logic design* stage, mediating between conceptual design and physical implementation. This was very effectively achieved by the Relational Model proposal, whereby first-normal form (1NF) *tables* are utilized as an *abstract data type* to be ultimately implemented by file hardware. And, to model table manipulation at the logic level, a *relational algebra* was defined [Codd].

Accordingly, we propose here a logic design stage for plot composition, involving an abstract structure for plots and a Plot Manipulation Algebra (PMA) to handle the structure, taking into due account the relations between narrative events implied by the conceptual level specification of the genre. To illustrate the discussion, as well as the design and use of a logic programming prototype tool, we employ an example involving a small number of events, which, in strikingly different combinations, have been treated repeatedly in literary works.

The paper is organized as follows. Section 2 covers the background for the present work. Section 3 gives a brief overview of PMA, whilst section 4 describes each operator. Section 5 deals, through a number of examples, with the prototype implementation. Section 6 contains concluding remarks.

2. Basic notions

2.1. Relations between events

To illustrate the event relations, we shall employ a simple example to be referenced along the paper. Consider four types of events, all having one woman and two men as protagonists: *abduction*, *elopement*, *rescue*, and *capture*. As demonstrated in folktale studies [Propp], many plots mainly consist of an act of villainy, i.e. of a violent action that breaks the initially stable and peaceful state of affairs, followed ultimately by an action of retaliation, which may or may not lead to a happy outcome.

Propp distinguished seven character roles (*dramatis personae*) according to the events assigned to each one's initiative: hero, villain, victim, dispatcher, donor, helper, false hero. Curiously, in literary texts involving the four events above, this distribution is not unique: we called the violent initial act "villainy", but the perpetrator of abduction, and more often of elopement, can be the hero of the narrative, and in such cases the woman's original guardian (husband, father) is regarded as the villain.

2.1.1. Syntagmatic Relations

To declare that it is legitimate to continue a plot containing abduction by placing rescue next to it, we say that these two events are connected by a *syntagmatic relation*. More precisely, we can define the semantics of the two events in a way that indicates that the occurrence of the first leaves the world in a state wherein the occurrence of the second is coherent. Similarly, a plot involving elopement followed by capture looks natural, and hence these two events are likewise related.

The syntagmatic relation between events induces a weak form of causality or enablement, which justifies their *sequential ordering* inside the plot.

2.1.2. Paradigmatic Relations

The events of abduction and elopement can be seen as *alternative* ways to accomplish a similar kind of villainy. Both achieve approximately – though not quite – the same effect: one man takes away a woman from where she is and starts to live in her company at some other place. There are differences, of course, since the woman's behaviour is usually said to be coerced in the case of abduction, but quite voluntary in the case of elopement. In fact, it is usual to assume that a sentence such as "Helen elopes with Paris", implies that Helen had fallen in love with Paris.

To express that abduction and elopement play a similar function, we say that there is a *paradigmatic relation* between the two events. Likewise, this type of relation is perceived to hold between the events of rescue and capture, which are alternative forms of retaliation. And, again, there is a difference between the woman's assumed attitude, associated as before with her feelings. An abducted woman expects to be rescued from the villain's captivity by the man she loves. On the contrary, she will only return through forceful capture if she freely eloped with the seducer.

As the present example suggests, the so-called syntagmatic and the paradigmatic axes [Saussure] are really *not* orthogonal in that the two relations cannot be considered independently when composing a plot. Thus, in principle, the two pairs enumerated in the previous section (abduction-rescue and elopement-capture) are the only normal ones, the former illustrated by the Sanskrit *Ramayana* [Valmiki] and the similarly structured Arthurian romance of *Lancelot* [Chrétien; Furtado & Veloso], and the latter by the Irish *Story of Deirdre* [McGarry]. Yet the next section shows that such limitations can, and even should, be waived occasionally.

2.1.3. Antithetic relations

While normal plots, whose outcome is fully determined, can be composed exclusively on the basis of the two preceding relations, the possibility to introduce unexpected turns is often desirable in order to make the plots more attractive – and this requires the construct that we chose to call *antithetic relation*. A context where a woman suffers abduction by a ravisher whom she does not love would seem incompatible with a capture event, since there should be no need to employ force to bring back the victim. So, in this sense, abduction and capture are in antithetic relation.

The mythical *Rape of the Sabines* shows what can happen as a consequence of a drastic reversal of the circumstances. King Romulus is facing a problem at the newly founded city of Rome: the population is entirely male at first. To remedy the lack, he leads his men to break into the dwellings of the Sabines and abduct their women. Sometime afterwards the Sabine warriors march against the Romans, but the women have no wish to be taken back, leaving to their countrymen no option except their capture. King Romulus's men had lawfully married them and made them bear children. A Roman chronicle [Titus Livius] reports the radical change in the women's feelings, and tells how the seemingly inevitable confrontation ended with the reconciliation of the two parties.

In contrast, modern history provides some distinctly regrettable examples of abduction actually followed by capture, categorized by psychiatrist Nils Bejerot as the *Stockholm syndrome*. One case in point is the abduction by a group of terrorists of the daughter of a millionaire, who ended up joining her tormentors in the practice of crimes, and was finally captured by the San Francisco police [Hearst & Moscow].

The occurrence of elopement followed by rescue provides a much stronger case of antithetic relation. Indeed, elopement only makes sense if the victim loves the seducer, whereas, for this very motive, she would resist to any attempt to rescue her, leaving forceful capture as the only viable alternative. Even so the legendary story of *Helen of Troy*, in spite of various discordant interpretations, seems to offer a counter-example. Married to king Menelaus of Sparta, Helen fled to Troy in the company of Paris, quite voluntarily according to a number of versions (e.g. the *Heroides* [Ovid]). But, after their escapade to Troy where they married, her love feelings started to wane while the Trojan War followed its bloody course and she kept recalling the far manlier Menelaus. The *Iliad* [Homer] signals repeatedly this critical change of sentiment. At the end her recovery turned from capture into rescue, as registered in the *Aeneid* [Virgil]. Paris was dead, and she had been delivered to Paris's brother Deiphobus. When the Greeks came out of the wooden horse and stormed the Trojan palaces, Helen herself made sure that Menelaus should win – and know that she was helping him in atonement for her previous misconduct. The shadow of Deiphobus tells

the episode to Aeneas; and what better example of irony could we find than his calling Helen "this peerless wife"?

One more example appears in the story of *Tristan and Isolde*, in several versions [Marchello-Nizia]. The knight had eloped with the queen; they were living in harsh conditions in a forest. The dramatic change of their love feelings, which allowed Isolde's rescue by king Mark to be achieved through a simple invitation, with no need to fight, had a very curious cause – the timely expiry date of the love potion they had drunk before, when sailing from Ireland to Cornwall [Béroul].

Generally speaking, if some *binary opposition* – the "to love or not to love" dilemma, in the present case – is allowed to be manipulated via some agency external to the predefined events, then one can have plots that no longer look conventional. A sort of discontinuity is produced by such radical shifts in the context. Intervening between abduction and capture, or between elopement and rescue, a sudden change of feelings can give rise to these surprising sequences. Also, both in fiction and in reality, things not always proceed according to planned events. Natural phenomena and disasters, the mere passage of time, the intervention of agents empowered to change the rules, supernatural or magic manifestations, etc., cannot be discounted.

Specifically for the tragedy genre, the *Poetics* [Aristotle] distinguishes between simple and complex plots, characterizing the latter by the occurrence of *recognition* (αναγνορισισ) and *reversal* (περιπετεια). Differently from reversal, recognition does not imply that the world itself has changed, but rather the *beliefs* of one or more characters about the actual facts. Because of a change of beliefs, a reason to be added to those enumerated in the previous paragraph, a reversal in the course of actions can take place, usually in a direction totally opposite to what was going on so far. Yet another possible external cause of both recognition and reversal in the tragic scene was the intervention of a god, who was lowered onto the stage using a crane, the so-called *deus ex machina*.

Aristotle's remarks are clearly relevant to the discussion of plots in general. Following his lead, we have admitted in a previous work [Ciarlini & al, 2008] state changes outside the regular regime of predefined events by allowing the user – literally acting *ex machina* (via the computer...) – to impose variations to the context (both in terms of *facts* and of *beliefs*), and thereby deviate the action from its predicted path.

This extreme device will be necessary to allow the elopement-rescue sequence. It may not be indispensable, however, for abduction-capture, if one wishes to leave room for *erroneous* beliefs (a case of Aristotle's ηαμαρτια), contradicting the actual facts. Criminal records everywhere are full of simulated abduction pacts for drawing a ransom from a deluded family. Conversely, a man can unnecessarily decide that capture is the only way to bring back a woman, if he mistakenly believes her to love the ravisher.

Figure 1 shows the relations thus far discussed.

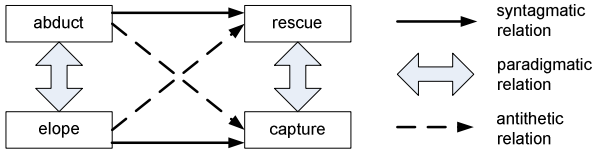

Figure 1: Syntagmatic, paradigmatic, and antithetic relations.

2.1.4. Meronymic relations

Meronymy is a word of Greek origin, used in linguistics to refer to the decomposition of a whole into its constituent parts. Forming an adjective from this noun, we shall call *meronymic relations* those that hold between an event and a lower-level set of events, with whose help it is possible to provide a more detailed account of the action on hand.

Thus, we could describe the abduction of a woman called Sita by a man called Ravana (characters taken from the *Ramayana* [Valmiki]) as: "Ravana rides from Lanka to forest. Ravana seizes Sita. Ravana carries Sita to Lanka." And her rescue by Rama could take the form: "Rama rides from palace to Lanka. Rama defeats Ravana. Rama entreats Sita. Rama carries Sita to palace." (Figure 2).

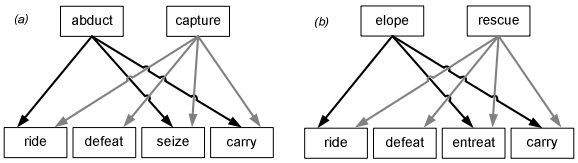

Figure 2: Meronymic relations:
(a) the forceful actions and (b) the gentle actions.

Detailing is most useful to pass from a somewhat abstract view of the plot to one, at a more concrete physical level, that is amenable (possibly after further decomposition stages) to the production of a computer graphics animation [Ciarlini et al., 2005]. Mixed plots, combining events of different levels, do also make sense, satisfying the option to represent some events more compactly while showing the others in detail.

2.2. Conceptual specification of genres

We believe that, in order to model a chosen genre, to which the plots to be composed should belong, one must specify at least:

a. what can exist at some state of the underlying mini-world,
b. how states can be changed, and
c. the factors driving the characters to act.

Accordingly, we start with a conceptual design method involving three schemas – static, dynamic and behavioural – which has been developed for modelling literary genres encompassing narratives with a high degree of regularity, such as fairy tales, and application domains of business information systems, such as banking, which are obviously constrained by providing a basically inflexible set of operations and, generally, by following strict and explicitly formulated rules [Furtado et al. 2008]. Indeed, in our model, we equate the notion of event with the state change resulting from the execution of a predefined operation.

The *static schema* specifies, in terms of the *Entity-Relationship* model [Batini et al.], the entity and relationship classes and their attributes. In our simple example, character and place are entities. The attributes of characters are name, which serves as identifier, and gender. Places have only one identifying attribute, pname. Characters are pair-wise related by relationships loves, held_by and consents_with. The last two can only hold between a female and a male character; thus held_by(Sita,Ravana) is a *fact* meaning that Sita is forcefully constrained by Ravana, whereas consents_with(Sita,Ravana) would indicate that Sita has voluntarily accepted Ravana's proposals. Two relationships associate characters with places: home and current_place. A state of the world consists of all facts about the existing entity instances and their properties holding at some instant.

The *dynamic schema* defines a fixed repertoire of operations for consistently performing state changes. The *STRIPS* [Fikes & Nilsson] model is used. Each operation is defined in terms of pre-conditions, which consist of conjunctions of positive and/or negative literals, and any number of post-conditions, consisting of facts to be asserted or retracted as the effect of executing the operation. Instances of facts such as home and gender, are fixed, not being affected by any operation. Of special interest are the *user-controlled* facts which, although also immune to operations, could be, as suggested in [Ciarlini et al, 2008], manipulated through arbitrary *directives*. In our example, loves is user-controlled.

Again for the present example, we have provided operations at two levels. The four main events are performed by level-1 operations: abduct, elope, rescue and capture. Operations at level-2 are actions of smaller granularity, in terms of which the level-1 operations can be detailed: ride, entreat, seize, defeat, and carry. Of course it would be possible to continue with this decomposing process, until a level of "primitive acts" is reached [Schank & Colby].

Our provisional version of the *behavioural schema* consists of goal-inference (a.k.a. situation-objective) rules, belief rules, and emotional condition rules.

For the example, three goal-inference rules are supplied. The first one refers to the ravisher. In words, in a situation where the princess is not at her home and the hero is not in her company – and hence she is unprotected – the ravisher will want to do whatever is adequate to bring her to his home. The other goal-inference rules refer to the hero, in two different situations having in common the fact that the ravisher has the woman in his home: either the hero believes that she does not love the other man, or he believes that she does. In both situations, he will want to bring her back, freely in the first case and constrained in the second.

Informally speaking, beliefs correspond to the partial view, not necessarily correct, that a character currently forms about the factual context (for a formal characterization, cf. the BDI model [Cohen & Levesque; Rao & Georgeff]). The belief rules that we formulated for our example look rational, but notice that they are treated as defaults, which could be overruled by a directive. A man (the hero or the ravisher) believes that the woman does *not* love his rival if the latter has her confined, but if she has ever been observed in his company and in no occasion (state) was physically constrained, the conclusion will be that she is consenting (an attitude seemingly too subjective to be ascertained directly in a real context).

The emotional condition rules refer to the three characters. A man (or woman) is happy if currently in the company of his (or her) beloved, and bored otherwise. A special condition applies to the woman: she will be *absolutely* happy if, in addition to the first motive for contentment, she has never been constrained by any of the two adversaries.

2.3. Revisiting the database relational algebra

Codd's *relational model* is widely recognized today as providing effective guidance for database design at the logical level. The abstract data type on which it is based is the *n-ary relation*, also known as (relational) *table*, defined as a subset of the Cartesian product of n data domains: $R \subseteq D1 \times D2 \times ... \times Dn$. The elements of relation R (or, equivalently, lines of table R) are called *tuples*. In other words, tables can be viewed as sets of tuples, a convenient abstraction for a hardware *file*, as tables are ultimately represented at the physical design level.

To handle database structures, Codd proposed [Codd] a *relational algebra* sublanguage, whose operator set can be reduced, as can be easily demonstrated [Ullman & Widom], to five primitives: *product*, *projection*, *union*, *selection* and *difference* (to which a sixth operator, *renaming*, is commonly added for practical convenience).

All operators have tables as operands and yield tables as their result. Product, union and difference are called binary operations (since they involve two operands), whereas projection and selection are unary operations (one operand).

Codd's database operators can be adapted for our present purposes if we substitute our plot abstract data type for tuples and, consequently, sets of plots (to be called *libraries*) for the relational tables. In practice, establishing this sort of analogy between database and narrative notions seems justified because the algebraic operators may indeed serve a similar purpose, associated in both cases with the first three event relations we have been talking about: syntagmatic, paradigmatic, antithetic, and also (as will be shown later) meronymic.

Starting with the syntagmatic dimension, one may observe that database product allows to put together two tables "horizontally", producing a result table whose tuples will be the concatenation of all pairs of operand tuples. One might say that the new table *extends* the information contained in the tuples of the first table, since it contains that information *and* information from additional domains provided by the second table. Projection on a given table, acting in an inverse direction, extracts from the operand tuples the elements pertaining to the indicated domains.

By contrast, turning to the paradigmatic dimension, union puts together two tables "vertically", in the sense that the resulting table can be seen as a copy of all tuples of one operand table followed by all tuples of the other (eventual duplicates being eliminated). So the new table provides *alternatives*, in the sense that it contains tuples that figure in the first *or* in the second table. Selection, on the other hand, allows to choose from a table those tuples whose values at the indicated column-positions meet a given requirement.

Codd formulated his algebra as a language mainly directed to model how database *queries* could be formulated. But it became clear that the other essential service of database manipulation, namely *updates*, could also be covered. In particular, *inserting* a tuple in a table can be expressed as the union of the table with a singleton table consisting of the tuple to be added. But how could we formulate the other kind of update, the process of *removing* a tuple from a table? The answer will serve to add further justification to difference, which is the relational algebra operation associated with the antithetic dimension, intuitively expressing a notion of negation. Generally speaking, the difference of two tables is a table containing those tuples in the first table that are *not* present in the second. In particular, a

removal update from a table can be formulated as the difference between the table and a singleton table consisting of the tuple to be deleted.

The meronymic dimension is not directly encompassed by the original version of relational algebra, because the formalism purported to handle first-normal form relations (1NF) (a.k.a. flat tables) exclusively. But implemented query languages based on the relational formalism, such as SQL, provided features that in fact went a step beyond these restrictions – such as group-by and a number of attendant aggregate numerical functions.

More fundamentally, complex types of data, describing for example assembled products or geographical units, characterized conceptually via a semantic part-of hierarchy [Smith & Smith], were found to require the addition of the so-called NF^2 (non first normal form) or nested tables at the logical level of design. To cope with this expansion, an extended relational algebra was required, including operators such as "partitioning" and "de-partitioning" [Furtado & Kerschberg], or "nest" and "unnest" [Jaeschke & Scheck] to convert from 1NF into NF^2 tables and vice-versa. Some domains of a tuple might in turn be a set of tuples, over domains pertaining to a finer-grained scale. Assume, for example, that a product p has components c1,c2,...,cn. In an NF^2 table, this part-of decomposition property would lead to a single tuple of the form <p,{c1,c2,...,cn}>, instead of being scattered throughout n separate <p,ci> tuples of a conventional 1NF table.

For our present work, we thought of operators for moving down or up in part-of hierarchies: factoring for replacing the whole by its parts, and combination for the inverse movement. In the example above, factoring applied to p would yield {c1,c2,...,cn}, whereas combination would yield p as corresponding to the given set of components. In the domain of natural numbers, incidentally, factoring corresponds to the decomposition of a number, which may proceed until its prime constituents are obtained.

3. Overview of PMA

3.1. The plot data structure

A *plot* P is a pair [S,D], where:

- S, the *event-set*, is a set of tagged events;
- D, the *dependency-set*, is a set of order dependencies, expressed as tag-pairs.

Tags are terms of the form fi, where i is a positive integer. By convention, the tags in an event-set S are numbered consecutively, starting with f1. Also by convention, the tagged events in S are required to be placed in a viable sequence, i.e. some sequence compatible with the partial order requirements expressed in D. Such conventions lead in general to simpler and more efficient algorithms to handle the data structure.

The order dependencies are determined exclusively on the basis of the satisfaction of post-conditions by pre-conditions. On determining the dependency-set D, another simplifying convention is adopted: any dependencies deducible by transitivity are omitted.

As an example, consider:

P = [[f1: ride(Ravana, Lanka, forest), f2: entreat(Ravana, Sita), f3: seize(Ravana, Sita))], [f1-f2,f1-f3]]

We say that two plots Pi and Pj are *similar* if, even with different parameter values and placed in a different (viable) sequence in the event-set, they involve the same:

- number and type of events
- order dependencies
- co-designation/ non-co-designation schemes

Co-designation (or, respectively, non-co-designation) allows (forbids) the occurrence of the same value at different parameter positions. Notice, for instance, that in the example above Sita occurs in the second position of seize and of carry; a plot with Guinevere in both places would meet the same co-designation requirement. To verify whether the order dependencies are the same, one looks for a renaming of the tags of one of the plots that can render the sets of order dependencies in the two plots identical. Two similar plots are said to be *equal* if the values at the corresponding parameter positions are identical, as happens with plot P shown above and a plot P' represented as follows:

P' = [[f1: ride(Ravana, Lanka, forest), f2: seize(Ravana, Sita), f3: entreat(Ravana, Sita))], [f1-f2,f1-f3]]

3.2. The operator set

The chosen algebraic operators should be sufficient to allow the composition of plots taking the four types of event relations into consideration. Our choice was guided by the analogy established with the database relational algebra in section 2.4.

Along the syntagmatic axis, plots consisting of one or more events are chained together to form longer sequences by the *product* operator. Inversely, a passage of interest can be extracted from a plot by the *projection* operator. Along the paradigmatic axis, the *union* operator offers different alternatives at the same position in the sequence. Libraries, as sets of plots, are created and expanded by applying union. To check if a plot has some desired characteristic, or which plots in a library do possess it, the *selection* operator is used.

Binary oppositions [Chandler] require the *difference* operator, one of whose purposes is to remove plots from a library.

These first five operators – product, projection, union, selection, difference – correspond to the five primitive operators of Codd's relational algebra. As mentioned earlier, the relational algebra can be said to be complete only if the database is restricted to flat tables, but its operator set is no longer sufficient if NF^2 tables are considered, typically to cope with part-whole schemes. For narratives, similarly, the meronymic relation between events, having to do with the desired level of granularity, led to the inclusion of the *factoring* and *combination* operators, respectively to detail or to summarize a plot.

The table below shows the entire operator set. Note that product, union and difference are binary operations, whereas projection, selection, factoring and combination are unary. All operators are applicable to both plots and libraries.

product	P1 * P2
projection	proj [T] @ P
union	P1 + P2

selection	sel [T]/E @ P
difference	P1 - P2
factoring	fac P
combination	comb P

3.3. Composing algebraic expressions

As expected, operators can be composed to form arbitrarily long expressions, whose evaluation will result either in a plot or in a library. The evaluation of an algebraic expression E, and the attribution of the result to a variable P, is indicated by writing:

P := E

The operands in E can either be plots or libraries represented explicitly, or sub-expressions at any depth. An operand may be represented as a variable, but at the moment it is to be handled by an operator it should be instantiated to some term expressing a plot or a library – see however how the case of a still uninstantiated variable is treated in our prototype implementation (example at section 5.3).

A notational shorthand was introduced for plots consisting of a single event. Take the event ride('Ravana', 'Lanka', forest), for example. As any other one-event plot it should be written as:

[[f1: ride(Ravana, Lanka, forest)], []]

but it is convenient to abbreviate that to a straightforward event syntax, i.e. ride(Ravana, Lanka, forest).

A slightly more involved expression is shown below (for its evaluation, cf. section 5.2). It illustrates the interplay of syntagmatic, paradigmatic and antithetic relations that we indicated for our example mini-world – all villainy-retaliation pairs will be formed, except elope followed by rescue:

P := (abduct(Ravana, Sita) + elope(Ravana, Sita)) * (rescue(Rama, Sita) + capture(Rama, Sita)) - elope(Ravana, Sita) * rescue(Rama, Sita)

4. Algebraic operators

4.1. Product

Given two plots P1 = [S1,D1] and P2 = [S2,D2], their product P := P1 * P2 is a plot P = [S,D], where S contains the S1 and the S2 events, and D contains the dependency-pairs to be computed anew between the S events, regardless of their provenance from S1 or S2.

Assuming that the event-sets S1 and S2 are organized in a viable sequence, the classic balance-line method to merge two sorted sets can be conveniently adapted, so as to obtain the resulting event-set S also with this property. One will recall that the method consists of,

at each step, comparing the first element of one set with the first of the other. The smallest of the two elements is inserted in the result and removed from its respective set, and the comparisons continue until at least one set is empty.

To adapt the method for plots, we must replace the notion of "smallness" involved in simple lexicographic value comparison by a criterion based on dependency between events. On a first approximation, if ei ∈ Si and ej ∈ Sj, we would say that ej < ei if ei depends on ej on account of pre-/post-conditions linkage, thus implying that ej should precede ei. However the problem is a little more complex: even if there is no dependency in either direction between ei and ej, there may exist in Sj an event ej' such that ei depends on ej' and ej' depends on ej (directly or transitively across Sj). If this is the case ei clearly also depends on ej, and ej must precede ei in the resulting S (it is easy to see that the other events in the chain up to ej' will also be inserted in S before ei as the execution continues). We considered that, whenever there is no dependency between the events being compared, it would seem natural to choose the event coming from the first operand (S1, if the expression has the form P := P1 * P2).

Once the event-set S is constructed, with consecutive tags f1, f2,.., fn, the dependency-set D can be readily obtained by examining the pre-/post-conditions connections between the S events. Note that in all resulting dependency-pairs fi-fj, fi will be lexicographically less than fj, as a consequence of the viable sequence property of S.

If one or both operands are (non-empty) libraries rather than plots, the result is a library containing the product of each plot taken from the first operand with each plot from the second, according to the standard Cartesian product definition. If one of the operands is the empty plot, denoted by [], the result of the product operation is the other operand, and thus [] behaves as the neutral element for product. The case of an empty library, rather than an empty plot, demanded an implementation decision; by analogy with the zero element in the algebra of numbers, it would be justifiable to determine that a failure should result whenever an empty library occurred as one or both operands. However we preferred to once again return the other operand as result, i.e. to regard this case as a frustrated attempt to extend plots instead of an error. This option is consistent with our decision to ambiguously denote both the empty plot and the empty list by [].

4.2. Projection

Given a plot P' = [S',D'], its projection P := proj [T] @ P' is a plot P = [S,D], where S only contains the events of S' specified in the projection-template T, ordered according to the position of the terms in T, and D only contains dependency pairs involving events placed in S.

In turn the projection-template T is a sequence of terms F:O, where F is a tag and O an event. F and/or O can be variables; if O is not a variable, some or all of its parameters can be variables. The S events receive new tags, as usual consecutive starting from f1. If one of the purposes of a projection, entailed by the sequential disposition of terms in T, is to re-order S', care must be taken to ensure that S will be a viable sequence.

To determine the dependency-pairs in D, one could simply ignore the D' original dependencies and examine the pre-/post-conditions between the retained S events. However we opted for a strategy that more strongly preserves the order information conveyed by D'.

Suppose D' contains the pairs fi'-fj' and fj'-fk'; assume further that, in view of T, the event tagged fj' was not chosen but fi' and fk' were, with new tags fi and fk respectively. But since fk' depended on fi' in D' (via the suppressed fj', or through a longer transitive chain), the pair fi-fk would be considered for insertion in D (and would be present except if found redundant in view of transitivity with respect to the other pairs assembled in D, as explained before).

If the operand is a library, the result is a library containing the projection of the plots of the operand library. Note however that, since libraries are sets, they cannot contain duplicates, which may arise as the consequence of a projection that suppresses events distinguishing two or more plots – and such duplicates are accordingly eliminated from the result. Locating plot duplicates is not as trivial as it would appear, as discussed in the next section where it is a particularly critical issue. If the projection fails for some reason, e.g. because the projection-template T referred to a tag or event that did not figure in S', the result will be the empty plot (or empty list) [] rather than an error.

4.3. Union

Given two operands U1 and U2, each of them either a plot or a library, their union U := U1 + U2 will always be a library containing all plots in U1 and U2, no two equal plots being retained.

Since plots are partially rather than totally ordered, testing plot equality can be somewhat costly. Given P1 = [S1,D1] and P2 = [S2,D2], one would keep S1 fixed and then check if some permutation S2' of the events in S2 is such that S1 and S2' are identical, except for the tags, which would themselves become identical after retagging the events in S2' according to the established convention. Once the new tags are assigned, one would finally compute anew the dependency-set D2' over the S2' events, and check if D1 and D2' were rendered identical, in which case P1 and P2 are in fact equal. Some expedients can alleviate the computational cost; if the cardinalities of S1 and S2 and also of D1 and D2 are not the same, then the two plots cannot be equal. The worst case occurs when D1 and D2 are empty (no ordering) and S1 and S2 contain the same number of events of the same type, in which situation there may be no way to avoid computing all permutations of S2.

One or both operands can be the empty library, ambiguously denoted as said before by []. If one of the operands is a plot and the other is [], their union is a library consisting of this single plot. As expected, the union of [] with itself is also [], and thus the empty library functions as the neutral element for union.

4.4. Selection

Given a plot P' = [S',D'], its selection P := sel [T]/E @ P' is the plot P' itself if the matching of the selection-template T against P' succeeds, as well as the subsequent evaluation of the logical expression E, also involving information taken from P'. The presence of expression E is optional, except when T is empty. If the test fails, the result to be assigned to P is the empty library [].

More often than not the operand of selection will be a library, and its result will also be a library, containing all plots that satisfy the test, or the empty library [] if none does. In order to select one plot at a time from a library L, even if L contains a single plot, one should use an expression of the form P := sel [T]/E @ one(L).

4.5. Difference

Given two operands U1 and U2, each of them either a plot or a library, their difference U := U1 - U2 will always be a library containing all plots in U1 that are not equal to any plot in U2, where plot equality is evaluated exactly as described for union. As happens with standard set difference, the result of the operation is not affected by plots in U2 that have no equals in U1.

If all plots in U1 have their equals in U2, the empty list [] is assigned to P.

4.6. Factoring

Given a plot P' = [S',D'], its factoring P := fac P' is a plot P = [S,D], where each level-1 event ei' present in S' is replaced by a sequence ei1, ei2,...,ein of level-2 events. These are obtained from a map declaration map(Ei',[Ei1,Ei2,...,Ein]), which must have been pre-defined, such that Ei' matches ei'.

In map declarations all terms in both arguments represent events, generally containing variables at the parameter positions. The first argument must be a level-1 event and the second a sequence of level-2 events.

When specifying a map declaration for an event Ei', care should be taken that the indicated sequence of level-2 operations should work as a plan successfully applicable at world situations satisfying the pre-conditions of Ei', and producing at the end the effects expressed by the post-conditions of Ei'. Also, if the sequence may have other (secondary) effects, these should not contradict those originally expected from the execution of Ei'.

As a map declaration for Ei' is found to match an event ei' in the course of factoring, all variables in Ei' will be instantiated with the values contained in the respective parameter positions of ei', with the consequence that several parameters of the level-2 events will also be instantiated by consistent variable substitution. In most cases, however, several level-2 events will not become fully ground terms.

In order to further instantiate the level-2 event parameters, we decided to apply a heuristic process based on the pre-condition declarations of these events. As said before, some database facts of the mini-world may be invariant, in the specific sense that none of the events provided may change them, an example being the gender of the acting characters. So, for example, if only one female character exists at the initial state, and the pre-condition of an event requires a female character at a certain parameter position, it seems natural to assign that character's name to the corresponding variable.

Once the events to be placed in S are thus obtained and instantiated as much as possible, they are tagged in the usual way and the dependency-set D is computed from the pre-/post-conditions of the events, a process that is facilitated by the consistent instantiation of the parameter positions.

If the operand is a library, the result is a library containing the factoring of all plots in the operand library.

4.7. Combination

Given a plot P' = [S',D'], its combination P := comb P' is a plot P = [S,D], where each sequence ei1', ei2',...,ein' of level-2 events present in S' is replaced by a level-1 event obtained by the inverse application of the pre-defined map declaration whose second argument happens to match the sequence.

To ensure the application of factoring it is sufficient to declare a map for each kind of level-1 event, as indicated in the previous section. For combination, however, the situation is far more complex, since level-2 operations involve a more detailed view of the mini-world. Certain unanticipated sequences may appear in plots, which may or may not have literary interest but which in any case, being unanticipated, would lack a pre-defined map declaration.

Another difficulty is that a sequence for which there is a map declaration may be interspersed with other extraneous events and would by this reason require a more complex matching process. And, even if duly detected and replaced via the combination operator, there would remain the tricky problem of where to place the extraneous events (except if projection were previously applied to determine the preferred position of such events).

If the operand is a library, the result is a library containing all plots in the operand library with the eventual modifications performed by applying the combination operation.

4.8. Extension: power and iteration

Two features extend the product operator, to achieve repeated plot sequences.

Given a plot P' = [S',D'], the nth power of P', expressed by P := P' ** N, for a non-negative integer N, is evaluated according to the recursive formula:

- if N = 0, P = []
- if N = 1, P = P'
- else P = P' * P' ** (N - 1)

Given a plot P' = [S',D'], the iteration of P', expressed by P := E@P', where E is a logical expression having any number of variables in common with P', is evaluated as follows:

- first, the iterator-template T is obtained, as the set of all possible instantiations of E at the initial state, and then:
- if T is { }, P = []
- else, if T = {t1, t2, ..., tn}, P = P'_{t1} * $P'_{\{t2, ..., tn\}}$

where P'_{ti} denotes P' with its variables instantiated consistently with those figuring in ti, and the subscrit in $P'_{\{ti+1, ..., tn\}}$ refers to the remaining instantiations of T to be be used at the next stages.

In particular, iteration can naturally be used to control the application of the power operation, since one of the effects of computing the iterator-template can be the instantiation of the exponent N, as in P := E@P' ** N.

As happens with product (and with the other basic operators), both features apply to single plots and to plot libraries.

4.9. Extension: patterning

Plot patterns can be obtained as a generalization of plots, by consistently substituting variables for the parameters and tags.

Given a plot P = [S,D], the pattern *P* of P, expressed by *P* := patt P, is obtained from P by substituting variables for the parameters and tags in both S and D. As expected, patterning applies to both single plots and to entire libraries.

To take advantage of this feature, we provided an additional version of the selection operator, in which a plot can be used as selection-template. Let P' = [S',D'] be a plot at an early phase of composition, still with very few events, and let L be a library representing a repository of typical narratives collected from diverse sources (for folktale narratives, see for instance [Aarne & Thompson]). Suppose further that the constituent plots of L have events pertaining to the same genre of P', though possibly defined over entirely different parameter values. If one wishes to extend P' to a set of fuller alternative plots containing all events in S' and preserving the order requirements imposed by D', it is possible to take as suggestions – or more precisely to re-use – the typical narratives in L by way of a pattern-matching technique. This is accomplished by evaluating the expression P := sel P'@ (patt L), wherein plot P' is used to conduct selection against the result of converting all plots in L into patterns. To obtain just one plot at a time, the expression P := sel P'@ one(patt L) can be employed.

5. A prototype implementation

A very simple PMA prototype was implemented to experiment with the notions discussed here. It serves to compose plots by applying the repertoire of algebraic operations, optionally resorting to pre-defined plots and plot libraries to supply useful clues. The entire system was written in SWI-Prolog[1], and comprises 3 modules:

1) The controlling module, called alg.pl, where the PMA operations are defined;
2) the example module, called sita_example.pl, which contains the conceptual specification of the example mini-world;
3) the plan-generator module, called warbeta.pl, containing, besides the plan-generation algorithm (an extended version of the early *Warplan* algorithm [Warren]), a number of routines for testing pre-conditions and post-conditions of the operations and for plan checking and simulated execution.

[1] http://www.swi-prolog.org/

The third module was developed as part of a previous work, in which the plan-generator module is the central mechanism, so that plot composition is mainly achieved through plan generation [Ciarlini & al, 2008]. For the present work, however, wherein plot composition arises from the manual formulation of algebraic expressions, the module plays an auxiliary role, its routines being called from inside the algebraic operators, especially for handling pre-conditions and post-conditions. Appendix I (reproducing a section of [Ciarlini & al, 2008]) describes some of its features; in particular, the use of different notations for formulating different kinds of pre- and post-conditions is a crucial aspect in the definition of the event-producing operations listed in Appendix II, where the complete conceptual schema and an initial state formulation are shown. Appendix III displays pre-defined plots and libraries, some of which are involved in the examples described next.

5.1. Example 1

Putting together two plots, via the product operator, cannot always take the form of a straightforward concatenation of the respective event sequences. The sequences may need to be merged in view of the partial order requirements.

Given the plots P1 and P2 below, consider their product P. As we shall repeat for the various examples, the result is shown both in plot format and in template-driven natural language.

```
P1:= carry('Ravana', 'Sita', 'Lanka') *
       ride('Rama', palace, 'Lanka') *
       defeat('Rama', 'Ravana'),
P2:= ride('Ravana', 'Lanka', forest) *
      seize('Rama','Sita') *
      carry('Rama','Sita',palace),
P := P1 * P2.
```

P = [[f1:ride(Ravana, Lanka, forest), f2:carry(Ravana, Sita, Lanka), f3:ride(Rama, palace, Lanka), f4:defeat(Rama, Ravana), f5:seize(Rama, Sita), f6:carry(Rama, Sita, palace)], [f1-f2, f2-f3, f3-f4, f4-f5, f5-f6]]

Ravana rides from Lanka to forest. Ravana carries Sita to Lanka. Rama rides from palace to Lanka. Rama defeats Ravana. Rama seizes Sita. Rama carries Sita to palace.

5.2. Example 2

The evaluation of plot P below illustrates the interplay between the syntagmatic, paradigmatic and antithetic event relations. The union operation is used to obtain a pair of libraries, consisting of the alternatives, respectively, for villainy and for retaliation. All villainy-retaliation sequences are then formed by computing the product of the two

libraries. Finally, via the difference operator, the transgressive sequence elope-rescue is excluded.

```
P := (abduct(V,W) + elope(V,W)) * (rescue(H,W) + capture(H,W)) -
      elope(V,W) * rescue(H,W).
```

P = [[[f1:abduct(Ravana, Sita), f2:rescue(Rama, Sita)], [f1-f2]], [[f1:abduct(Ravana, Sita), f2:capture(Rama, Sita)], [f1-f2]], [[f1:elope(Ravana, Sita), f2:capture(Rama, Sita)], [f1-f2]]]

Ravana abducts Sita. Rama rescues Sita.

Ravana abducts Sita. Rama captures Sita.

Ravana elopes with Sita. Rama captures Sita.

5.3. Example 3

This example serves to illustrate two points. Firstly, it shows how should be treated a variable, such as E in the expression below, if it is still uninstantiated at the moment when the evaluation process would take it as operand. The tool regards it as a slot to be filled in with a chosen event, and engages the user in a menu-driven dialogue. Secondly, this is in a sense a non-story example: no villainy or retaliation occurs; the hero simply goes fetch the princess in the forest and brings her back to their home. The choice (by instantiating variable E) is whether persuasion (an entreat event) or force (seize) will be employed.

```
P :=ride('Rama', palace, forest) *
     E *
     carry('Rama', 'Sita', palace).
```

options:
1:abduct
2:elope
3:rescue
4:capture
5:ride
6:seize
7:entreat
8:carry
9:defeat

event? - 7.
entreat(A, B)

[[f1:entreat(A, B)], []]

```
for A, choose number from:
1:Sita
2:Rama
3:Ravana
4:forest
5:palace
6:Lanka
A goes to? - 2.

for B, choose number from:
1:Sita
2:Rama
3:Ravana
4:forest
5:palace
6:Lanka
B goes to? - 1.

event: [[f1:entreat(Rama, Sita)], []]

P = [[f1:ride(Rama, palace, forest), f2:entreat(Rama, Sita), f3:carry(Rama, Sita, palace)],
      [f1-f2, f2-f3]]
```

Rama rides from palace to forest. Rama entreats Sita. Rama carries Sita to palace.

5.4. Example 4

Projection can be used to re-order the events in a plot. Given a pre-defined fake abduction plot, a new plot P can be obtained, preserving all the original events but inverting the position of the third and forth events. Instead of fake abduction, we now have a situation in which the villain initially acts as a seducer but, after having brought the princess to his home, decides to keep her in strict confinement.

```
fake_abduct(V,W,F) :-
    F := [[f1:ride(V,P1,P2), f2:entreat(V,W), f3:seize(V,W), f4:carry(V,W,P1)],
          [f1-f2,f1-f3,f2-f4]].

P := proj [f1:_,f2:_,f4:_,f3:_] @ fake_abduct('Ravana','Sita').

P = [[f1:ride(Ravana, Lanka, forest), f2:entreat(Ravana, Sita), f3:carry(Ravana, Sita,
Lanka), f4:seize(Ravana, Sita)], [f1-f2, f2-f3, f3-f4]
```

Ravana rides from Lanka to forest. Ravana entreats Sita. Ravana carries Sita to Lanka. Ravana seizes Sita.

5.5. Example 5

Using selection over the detailed description of the two level-1 villainy events, obtained through the factoring operation, we may determine which one involves the violent seize level-2 event. Note that the desired level-1 event is reconstituted from the selected level-2 description by applying the combination operator at the end.

P := comb sel [_:seize(_,_)] @ (fac (abduct(V,W) + elope(V,W))).

[[[f1:abduct(Ravana, Sita)], []]]

Ravana abducts Sita.

5.6. Example 6

Consider a plot P that starts with abduction, and should continue with an adequate form of retaliation. Choosing this second event can be done by performing a union of the two possible alternatives, making the effective presence of each of them dependent on a condition: the hero can only rescue the woman if she loves him, otherwise he must capture her. Note that this conditional union works as an exclusive-or or as an if-then-else scheme.

P := abduct(V,W) *
((sel []/loves(W,H) @ rescue(H,W)) + (sel []/(not loves(W,H)) @ capture(H,W))).

P = [[[f1:abduct(Ravana, Sita), f2:rescue(Rama, Sita)], [f1-f2]]]

Ravana abducts Sita. Rama rescues Sita.

5.7. Example 7

The level-2 events entreat and seize are the only forms of persuasion that the male characters may employ when dealing with the princess. Here, iteration is applied to make the two men successively confront Sita, taking her feelings with respect to them into consideration when deciding how to act. To a man loved by Sita it is enough to entreat her once, whereas an unloved man would have no alternative except seizing her (an action to be performed twice, as indicated by appropriately setting the power operator). Notice the use of an explicit if construct, provided by the tool as an auxiliary facility.

P := iter

```
      (gender(M,male),
        if(loves('Sita',M),
        (O=entreat(M,'Sita'),N=1),
        (O=seize(M,'Sita'),N=2)))  @
    ride(M,_,forest) * O ** N.
```

Suppose Sita loves the hero, Rama, but not the villain, Ravana. Then the logical expression controlling the iteration operator will evaluate to the iterator-template T as shown below. Note that T is represented as a list with two items containing as expected two different instantiations for the three variables M, O, and N.

T = [(M=Rama, O=entreat(Rama, Sita), N=1), (M=Ravana, O=seize(Ravana, Sita), N=2)]

P = [[f1:ride(Rama, palace, forest), f2:entreat(Rama, Sita), f3:ride(Ravana, Lanka, forest), f4:seize(Ravana, Sita), f5:seize(Ravana, Sita)], [f1-f2, f3-f4, f3-f5]]

Rama rides from palace to forest. Rama entreats Sita. Ravana rides from Lanka to forest. Ravana seizes Sita. Ravana seizes Sita.

5.8. Example 8

The pre-defined library lib_2, listed in Appendix II and repeated below together with its rendering in natural language for ease of reference, will be used to extend an initial plot P, by applying the special selection option against the pattern-converted library.

```
lib_2(
 [ [[f1:ride('Meleagant','Gore',forest),
     f2:seize('Meleagant','Guinevere'),
     f3:carry('Meleagant','Guinevere','Gore'),
     f4:ride('Lancelot','Camelot','Gore'),
     f5:defeat('Lancelot','Meleagant'),
     f6:entreat('Lancelot','Guinevere'),
     f7:carry('Lancelot','Guinevere','Camelot')],
     [f1-f2,f2-f3,f3-f4,f4-f5,f5-f6,f6-f7]],
   [[f1:ride('Tristan',forest,garden),
     f2:entreat('Tristan','Isolde'),
     f3:carry('Tristan','Isolde',forest),
     f4:ride('Mark','Cornwall',forest),
     f5:defeat('Mark','Tristan'),
     f6:seize('Mark','Isolde'),
     f7:carry('Mark','Isolde','Cornwall')],
     [f1-f2,f2-f3,f3-f4,f4-f5,f5-f6,f6-f7]]
     ]).
```

Meleagant rides from Gore to forest. Meleagant seizes Guinevere. Meleagant carries Guinevere to Gore. Lancelot rides from Camelot to Gore. Lancelot defeats Meleagant. Lancelot entreats Guinevere. Lancelot carries Guinevere to Camelot.

Tristan rides from forest to garden. Tristan entreats Isolde. Tristan carries Isolde to forest. Mark rides from Cornwall to forest. Mark defeats Tristan. Mark seizes Isolde. Mark carries Isolde to Cornwall.

If P is formulated as shown next, the Meleagant plot will serve as model to obtain the extended plot Pe. Notice the subsequent application of the combination operator to Pe in order to recognize the kind of narrative obtained, which turns out to be of the abduct-rescue variety.

```
P = [[f1:carry('Ravana','Sita','Lanka'),
       f2:entreat('Rama','Sita'),
       f3:carry('Rama','Sita',palace)],
     [f1-f2,f2-f3]],
Pe := sel P @ (patt lib_2),
Pc := comb Pe.
```

Ravana carries Sita to Lanka. Rama entreats Sita. Rama carries Sita to palace.

Pe = [[[f1:ride(Ravana, Lanka, forest), f2:seize(Ravana, Sita), f3:carry(Ravana, Sita, Lanka), f4:ride(Rama, palace, Lanka), f5:defeat(Rama, Ravana), f6:entreat(Rama, Sita), f7:carry(Rama, Sita, palace)], [f1-f2, f2-f3, f3-f4, f4-f5, f5-f6, f6-f7]]]

Ravana rides from Lanka to forest. Ravana seizes Sita. Ravana carries Sita to Lanka. Rama rides from palace to Lanka. Rama defeats Ravana. Rama entreats Sita. Rama carries Sita to palace.

Pc = [[[f1:abduct(Ravana, Sita), f2:rescue(Rama, Sita)], [f1-f2]]]

Ravana abducts Sita. Rama rescues Sita.

A different result is obtained by applying, to the same library, a different initial plot P which only diverges from the previous formulation with respect to the second event (seize, instead of entreat). Now the Tristan plot will be taken as model. The resulting plot Pe is finally recognized to be of the elope-capture variety, again by applying the combination operator to Pe.

```
P = [[f1:carry('Ravana','Sita','Lanka'),
       f2:seize('Rama','Sita'),
       f3:carry('Rama','Sita',palace)],
     [f1-f2,f2-f3]],
Pe := sel P @ (patt lib_2),
Pc := comb Pe.
```

Ravana carries Sita to Lanka. Rama seizes Sita. Rama carries Sita to palace.

Pe = [[[f1:ride(Ravana, Lanka, forest), f2:entreat(Ravana, Sita), f3:carry(Ravana, Sita, Lanka), f4:ride(Rama, palace, Lanka), f5:defeat(Rama, Ravana), f6:seize(Rama, Sita), f7:carry(Rama, Sita, palace)], [f1-f2, f2-f3, f3-f4, f4-f5, f5-f6, f6-f7]]]

Ravana rides from Lanka to forest. Ravana entreats Sita. Ravana carries Sita to Lanka. Rama rides from palace to Lanka. Rama defeats Ravana. Rama seizes Sita. Rama carries Sita to palace.

Pc = [[[f1:elope(Ravana, Sita), f2:capture(Rama, Sita)], [f1-f2]]]

Ravana elopes with Sita. Rama captures Sita.

6. Concluding remarks

The main contribution to be expected from a logic model is to provide reliable guidelines for developing practical systems that may be regarded as "complete" according to some criterion. We claim that PMA is complete in the specific sense that it covers plot manipulation along the four dimensions induced by the syntagmatic, paradigmatic antithetic and meronymic event relations. These relations, as we argued before [Ciarlini et al, 2008] seem to encompass some fundamental aspects of plot composition, associated in turn with the four major tropes of semiotic research [Booth; Chandler].

The direct use of PMA is not recommended, a remark applicable in fact to any logic level formalism, which, as said before, is nothing but an intermediate stage towards a computer-based user environment. In the database realm, relational algebra served as a well-fundamented basis for DBMS products, but its formulas remained hidden from common users behind user-friendly query languages. And notice that user-friendliness is not enough: some "intelligence" is needed to avoid meaningless commands, involving for instance an equality join based on the comparison of semantically unrelated fields.

Similarly, an effective system to help prospective authors should offer a friendly menu-based interface on top of the algebraic engine, able to offer opportune guidance – or at least helpful clues – at each composition stage. More importantly the system should have access to an online representation of the three-level conceptual schema specification, using this meta-level information to keep checking (through the plan-generation routines) the *semantic correction* of the plots being generated. Moreover, access to the behavioural schema in particular should serve to check what might be called *pragmatic plausibility*, i.e. whether the events caused by each character reflect the character's expected way of reacting, especially in view of the declared situation-goal rules.

At least two general limitations of our approach must be recognized, which must not be attributed to PMA, arising as they do from the conceptual model adopted:

- Since the dynamic schema restricts the narrative events to those resulting from a fixed repertoire of operations, only highly repetitive genres can be modelled;

- a consistent definition of an entire repertoire of operations by pre-/ post-conditions is hard to elaborate, on account of their complex mutual interferences, and may require a time-consuming series of trial-and-error adjustments.

With respect to PMA itself, much remains to be done. Theoretic work is needed to systematically investigate the formal properties of the algebra. Practical work, drawing from our early experiments with the **PlotBoard** prototype [Ciarlini et al, 2008], should include the design of friendly systems based on PMA, taking maximum advantage of the previously developed plan-generation facilities, but also allowing effective user interaction along a step-wise plot composition and adaptation process. Different initial states might be chosen to start such process, subsequent states being reached by simulated execution of the event-producing operations. Authors might ask to be constantly warned about the risk of correctness and plausibility breaches, but, nevertheless, should retain the power to introduce transgressive changes via user directives, as in **PlotBoard**, or some other semi-automatic intervention mechanism.

References

AARNE, A. AND THOMPSON, S. (1987). *The Types of the Folktale*. Suomalainen Tiedeakatemia.

ARISTOTLE (2000). "Poetics". In *Classical Literary Criticism*. Penelope Murray et al. (trans.). Penguin.

BAL., M. (2002). *Narratology*. U. of Toronto Press.

BATINI, C., CERI, S. AND NAVATHE, S. (1992). Conceptual Design – an Entity-Relationship Approach. Benjamin Cummings.

BÉROUL (1970). *The Romance of Tristan*. A.S. Fedrick (trans.). Penguin.

BOOTH, W. (1974). *A rhetoric of Tropes*. U. of Chicago Press.

CHANDLER, D. (2007). *Semiotics: The Basics*. Routledge.

CHRÉTIEN DE TROYES (1983). *Le Chevalier de la Charrete*. M. Rocques (ed.). Honoré Champion.

CIARLINI, A.E.M., POZZER, C.T., FURTADO, A.L. AND FEIJÓ, B. (2005). "A logic-based tool for interactive generation and dramatization of stories". In *Proc. of Advances in Computer Entertainment Technology*.

CIARLINI, A.E.M., BARBOSA, S.D.J., CASANOVA, M.A., FURTADO, A.L. (2008). "Event Relations in Plan-Based Plot Composition". In *Proc. of SBGames*.

CODD, E. F. (1972). "Relational completeness of data base sublanguages". In *Database Systems*, R. Rustin, ed., Prentice-Hall.

COHEN, P.R. AND LEVESQUE, H.J. (1990). "Intention is Choice with Commitment". *Artificial Intelligence* 42.

FIKES, R.E. AND NILSSON, N.J. (1971). "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence* 2.

FURTADO, A.L., CASANOVA, M.A., BARBOSA, S.D.J. AND BREITMAN, K.K. (2008). "Analysis and Reuse of Plots using Similarity and Analogy". In *Proc. of 27th International Conference on Conceptual Modeling (ER)*.

FURTADO, A.L. AND KERSCHBERG, L. (1977). "An algebra of quotient relations". In Proc. *ACM SIGMOD International Conference on Management of Data*.

FURTADO, A.L. AND VELOSO, P.A.S. (1996). "Folklore and Myth in The Knight of the Cart". In *Arthuriana*, vol 6, 2.

HEARST, P.C. AND MOSCOW, A. (1988). *Patty Hearst: her own Story*. Avon.

HOMER (1950). *The Iliad*. E.V. Rieu (trans.). Penguin.

JAESCHKE, G., SCHECK, H.J. (1982). "Remarks on the algebra of non first normal form relations". In *Proc. of the 1st ACM SIGACT-SIGMOD symposium on principles of database systems*.

LLOYD, W. (1987). *Foundations of Logic Programming*. Springer.

MARCHELLO-NIZIA, C. (org.) (1995). *Tristan et Yseut*. Gallimard.

MCGARRY, M. (ed.) (1979). "The Story of Deirdre". In *Great Folk Tales of Ireland*. Frederick Muller.

OVID (1986). *Heroides and Amores*. G. Showerman (trans.). Harvard U. Press.

PROPP, V. (1968). *Morphology of the Folktale*. S. Laurence (trans.). U. of Texas Press .

RAO, A.S. AND GEORGEFF, M.P. (1991). "Modeling rational agents within a BDI-architecture". In Proc. of *Int'l Conf. on Principles of Knowledge Representation and Reasoning*.

SAUSSURE, F. (2006). *Cours de Linguistique Générale*. C. Bally, A. and A. Riedlinger (eds.). Payot.

SCHANK, R.C. AND COLBY, K. (eds.) (1973). *Computer Models of Thought and Language*. W.H. Freeman.

SMITH, J.M. AND SMITH, D.C.P. (1977). "Data abstraction: aggregation and generalization". In *ACM Transactions on Database System*s, 2:2.

TITUS LIVIUS (1919). *History of Rome*, vol I. B. O. Fster (trans.). Harvard U. Press. Book I, Chapter I: 13.

ULLMAN, J. D. AND WIDOM, J. (2008). *A first Course on Database Systems*. Prentice-Hall.

VALMIKI (1999). *Le Ramayana*. Philippe Benoît et al. (trans.). Gallimard,.

VIRGIL (1994). *Eclogues, Georgics, Aeneid*. H.R. Fairclough (trans.). Harvard U. Press.

WARREN, D.H.D. (1974). *WARPLAN: a System for Generating Plans*. Edinburgh: University of Edinburgh, Department of Computational Logic, memo 76.

Appendix I - some features of the plan-generator

The plan generator follows a backward chaining strategy. For a fact `F` (or `not F`) that is part of a given goal, it checks whether it is already true (or false) at the current state. If it is not, it looks for an operation `Op` declared to add (or delete) the fact as part of its effects. Having found such operation, it then checks whether the pre-condition `Pr` of `Op` currently holds – if not, it tries, recursively, to satisfy `Pr`. Moreover, the plan generator must consider the so-called frame problem [Lloyd], by establishing (in second-order logic notation) that the facts holding just before `Op` is executed stay valid unless explicitly declared to be altered as part of the effects of `Op`.

Like goals, pre-conditions are denoted by conjunctions of literals and arbitrary logical expressions. We distinguish, and treat differently, three cases for the involved positive or negative facts:

a. facts which, in case of failure, should be treated as goals to be tried recursively by the plan generator;
b. facts to be tested immediately before the execution of the operation, but which will not be treated as goals in case of failure: if they fail the operation simply cannot be applied;
c. facts that are not declared as added or deleted by any of the predefined operations.

Note that the general format of a pre-condition clause is `precond(Op, Pr) :- B`. In cases (a) and (b), a fact `F` (or `not F`) must figure in `Pr`, with the distinction that the barred notation `/F` (or `/(not F)`) will be used in case (b). Case (c) is handled in a particularly efficient way. Since it refers to facts that are invariant with respect to the operations, such facts are included in the body `B` of the clause, being simply tested against the current state when the clause is selected.

An example is the precondition clause of operation `seize(M,W)`, where `M` is the agent and `W` the patient of the action. Clearly the two characters should be together at the same place, and, accordingly, the `Pr` argument shows two terms containing the same variable `P` to express this requirement, but the term for `W` is barred: `/current_place(W, P)`, which does not happen in `M`'s case. The difference has an intuitive justification: the prospective agent has to go to the place where the patient is, but the latter will just happen to be there for some other reason.

The proper treatment of (a) and (b) is somewhat tricky. Suppose the pre-condition `Pr` of operation `Op` is tested at a state S_1. If it fails, the terms belonging to case (a) will cause a recursive call whereby one or more additional operations will be inserted so as to move from S_1 to a state S_2 where `Op` itself can be included. It is only at S_2, not at S_1, that the barred terms in case (b) ought to be tested, and so the test must be *delayed* until the return from the recursive call, when the plan sequence reaching S_2 will be fully instantiated.

Operations can admit more than one precondition clause, so as to cope with different circumstances. This happens with the `carry(M,W,P2)` operation, whereby `W` will either freely consent to be transported to `P2` by `M`, or will have to be forcefully held by him.

With respect to the added and deleted clauses declaring effects of operations, the plan generator also employs a barred notation, to distinguish between two cases: (a) primary effects, and (b) secondary unessential effects. In case (a), if any fact `F` to be added by `Op` already holds, or already does not hold if it should be deleted, then `Op` is considered *non-productive* and fails to be included in the plan. In contrast, in case (b), such lack of effect would be admitted and cause no failure.

As an example, consider the clause of operation `capture(M1,W)` that declares as deleted the fact `held_by(W,M2)`, as a result of `M1`'s action to take away `W` from `M2`. Notice that the fact may or may not hold prior to capture; it will hold if `W` was abducted by `M2`, but will not hold if an elopement occurred instead – and that is why the barred notation is used for this particular deleted clause. On the contrary, the fact `current_place(W,P2)`, where `P2` is the home of `M2`, must necessarily be deleted by an effective execution of the operation, and so does not figure as barred.

The execution of plans is done through `assert` or `retract` commands on the facts to be, respectively, added or deleted. The plan's pre- and post-conditions are checked during the process, there being no effect in case of failure. A `log(L)` literal, initiated with `L=start`, is extended with each successful plan execution and can be usefully retrieved for a variety of purposes. On the basis of the log and of the initial state, which is saved when a session begins, it is possible to query about facts at any intermediate state. It is also possible to save and restore any previous state S (initial or intermediate), which enables simulation runs.

User interventions, necessary to achieve unplanned situations, are permitted in a limited scale through *directives* that can be either intermixed with the operations in a plan or called separately. Two of these are

used in our example, one for changing `loves` facts, immune to the predefined operations, and the characters' beliefs, which may not correspond to actual facts.

To finish this partial review of the plan features, we remark that the planning algorithm `plans(G,P)` is called in more than one way. More frequently `G` is given, as the goal, and `P` is a variable to which a generated plan will be assigned as output. However an inverse usage has been provided, wherein `P` is given and `G` is a variable; in this case, the algorithm will check whether `P` is valid and, if so, assign its net effects (a conjunction of `F` and `not F` terms) to `G`.

Appendix II - conceptual schemas and example initial state

```
:- dynamic believes/2,loves/2,
   current_place/2,
   held_by/2,
   consents_with/2,
   op_level/1 .

template(not F,['not true: '|Ft]) :-
  template(F,Ft).


% STATIC SCHEMA

entity(character,name).
attribute(character,gender).
entity(place,pname).
relationship(loves,[character,character]).
relationship(home,[character,place]).
relationship(current_place,[character,place]).
relationship(held_by,[character,character]).
relationship(consents_with,[character,character]).

template(character(X),['character ',X]).
template(gender(X,G),[X,' is a ',G1]) :-
  (G == male, G1 = 'man'; G == female, G1 = 'woman').
template(place(X),['place ',X]).
template(loves(X,Y),[X,' loves ',Y]).
template(home(X,L),['home of ',X,': ',L]).
template(current_place(X,L),['Current location of ',X,': ',L]).
template(held_by(W,M),[W,' is being held by ',M]).
template(consents_with(W,M),[W,' consents with ',M]).


user_controlled(loves(_,_)).


% DYNAMIC SCHEMA

added(X,Y) :- /added(X,Y).
deleted(X,Y) :- /deleted(X,Y).


% first level operators

operation(abduct(M2,W),1).
```

```
parm_type(abduct(M2,W),[character,character]).
deleted(current_place(W,Pc),abduct(M2,W)) :-
  home(M2,P2), not (Pc == P2).
/deleted(current_place(M2,Pc),abduct(M2,W)) :-

  home(M2,P2), not (Pc == P2).
added(current_place(W,P2),abduct(M2,W)) :- home(M2,P2).
/added(current_place(M2,P2),abduct(M2,W)) :- home(M2,P2).
added(held_by(W,M2),abduct(M2,W)).
precond(abduct(M2,W),current_place(W,Pc)) :-
   loves(M2,W),
   gender(W,female),
   home(M2,P2),
   home(W,P1),
   not (P1 == P2),
   place(Pc),
   not (Pc == P1),
   not (Pc == P2).
template(abduct(M2,W),[M2,' abducts ',W]).

operation(elope(M2,W),1).
parm_type(elope(M2,W),[character,character]).
deleted(current_place(W,Pc),elope(M2,W)) :-
  home(M2,P2), not (Pc == P2).
/deleted(current_place(M2,Pc),elope(M2,W)) :-
  home(M2,P2), not (Pc == P2).
added(current_place(W,P2),elope(M2,W)) :- home(M2,P2).
/added(current_place(M2,P2),elope(M2,W)) :- home(M2,P2).
added(consents_with(W,M2),elope(M2,W)).
precond(elope(M2,W),current_place(W,Pc)) :-
   loves(M2,W),
   loves(W,M2),
   gender(W,female),
   home(M2,P2),
   home(W,P1),
   not (P1 == P2),
   place(Pc),
   not (W == M2),
   not (Pc == P1),
   not (Pc == P2).
template(elope(M2,W),[M2,' elopes with ',W]).

operation(rescue(M1,W),1).
parm_type(rescue(M1,W),[character,character]).
deleted(current_place(W,P2),rescue(M1,W)) :-
  home(W,P1), not (P2 == P1).
/deleted(current_place(M1,P2),rescue(M1,W)) :-
  home(W,P1), not (P2 == P1).
/deleted(held_by(W,M2),rescue(M1,W)) :-
  home(W,P1), home(M2,P2), not (M2 == M1), not (P2 == P1).
added(current_place(W,P1),rescue(M1,W)) :- home(W,P1).
/added(current_place(M1,P1),rescue(M1,W)) :- home(W,P1).
added(consents_with(W,M1),rescue(M1,W)).
precond(rescue(M1,W),(current_place(W,P2), /(not held_by(W,M1)))) :-
   gender(W,female),
   home(W,P1),
   home(M2,P2),
```

```
   not loves(W,M2),
   character(M1),
   not (M1 == W),
   not (M1 == M2),
   not (P1 == P2).
template(rescue(M1,W),[M1,' rescues ',W]).

operation(capture(M1,W),1).
parm_type(capture(M1,W),[character,character]).
deleted(current_place(W,P2),capture(M1,W)) :-
  home(W,P1), not (P2 == P1).
/deleted(current_place(M1,P2),capture(M1,W)) :-
  home(W,P1), not (P2 == P1).
/deleted(held_by(W,M2),capture(M1,W)) :-
  home(W,P1), home(M2,P2), not (P2 == P1).
added(held_by(W,M1),capture(M1,W)).
added(current_place(W,P1),capture(M1,W)) :- home(W,P1).
/added(current_place(M1,P1),capture(M1,W)) :- home(W,P1).
precond(capture(M1,W),current_place(W,P2)) :-
   gender(W,female),
   home(W,P1),
   home(M2,P2),
   character(M1),
   not (M1 == W),
   not (M1 == M2),
   not (P1 == P2).
template(capture(M1,W),[M1,' captures ',W]).

template(directive(D),[D,' directive']).

% second level operators

operation(ride(C,P1,P2),2).
parm_type(ride(C,P1,P2),[character,place,place]).
deleted(current_place(C,P1),ride(C,P1,P2)) :- not (P1 == P2).
added(current_place(C,P2),ride(C,P1,P2)) :- not (P1 == P2).
precond(ride(C,P1,P2),current_place(W,P2)) :-
  loves(C,W),
  gender(C,male),
  gender(W,female),
  (home(C,P1); not home(C,P1),place(P1)),
  place(P1),
  place(P2),
  not (P1 == P2).
template(ride(C,P1,P2),[C,' rides from ',P1,' to ',P2]).

operation(seize(M,W),2).
parm_type(seize(M,W),[character,character]).
added(held_by(W,M),seize(M,W)).
precond(seize(M,W),
  (/current_place(W,P),
   current_place(M,P),
   not held_by(W,M2))) :-
  gender(M,male),
  gender(W,female),
  gender(M2,male),
  not (M == M2),
```

```
  place(P),
  not home(W,P),
  not home(M,P).
precond(seize(M,W),
  (/current_place(W,P),
   /current_place(M,P),
   /consents_with(W,M),
   /(not held_by(W,M2)))) :-
  gender(M,male),
  gender(W,female),
  gender(M2,male),
  not (M == M2),
  place(P),
  not home(W,P),
  home(M,P).
template(seize(M,W),[M,' seizes ',W]).

operation(entreat(M,W),2).
parm_type(entreat(M,W),[character,character]).
added(consents_with(W,M),entreat(M,W)).
precond(entreat(M,W),
  (/current_place(W,P),
   current_place(M,P),
   /(not held_by(W,M)),
   not held_by(W,M2))) :-
  loves(M,W),
  loves(W,M),
  gender(M,male),
  gender(W,female),
  gender(M2,male),
  not (M == M2),
  place(P),
  not home(M,P),
  not home(W,P).
template(entreat(M,W),[M,' entreats ',W]).

operation(carry(M,W,P2),2).
parm_type(carry(M,W,P2),[character,character,place]).
deleted(current_place(M,P1),carry(M,W,P2)) :-
  home(M,P2), not (P1 == P2).
deleted(current_place(W,P1),carry(M,W,P2)) :-
  home(M,P2), not (P1 == P2).
added(current_place(M,P2),carry(M,W,P2)) :-
  home(M,P2).
added(current_place(W,P2),carry(M,W,P2)) :-
  home(M,P2).
precond(carry(M,W,P2),
  (consents_with(W,M),
  /current_place(W,P1),
  /current_place(M,P1))) :-
  gender(W,female),
  gender(M,male),
  loves(M,W),
  loves(W,M),
  home(M,P2).
precond(carry(M,W,P2),
  (held_by(W,M),
```

```
  /current_place(W,P1),
  /current_place(M,P1))) :-
  gender(W,female),
  gender(M,male),
  loves(M,W),
  home(M,P2).
template(carry(M,W,P2),[M,' carries ',W,' to ',P2]).

operation(defeat(M1,M2),2).
parm_type(defeat(M1,M2),[character,character]).
deleted(held_by(W,M2),defeat(M1,M2)).
precond(defeat(M1,M2),
  (/current_place(M2,P),
   /current_place(W,P),
   /held_by(W,M2),
   /current_place(M1,P))) :-
   gender(M1,male),
   gender(M2,male),
   not (M1 == M2),
   loves(M1,W),
   place(P).
template(defeat(M1,M2),[M1,' defeats ',M2]).

img_template(T,T1) :-
  T =.. [O,'Ravana'|P],
  on(O,[ride,entreat,seize,carry]),!,
  concat(O,1,O1),
  T1 =.. [O1,'Ravana'|P].
img_template(T,T).


% BEHAVIOURAL SCHEMA

% goal-inference rules

sit_obj(M2,
  (current_place(W,P3),not current_place(M1,P3)),
  (current_place(W,P2))) :-
  gender(M1,male),
  gender(M2,male),
  gender(W,female),
  not (M1 == M2),
  place(P3),
  not home(P3,_),
  home(W,P1),
  home(M2,P2),
  not (P1 == P2).

sit_obj(M1,
  (current_place(W,P2), believes(M1,not loves(W,M2))),
  (current_place(W,P1), not held_by(W,M1))) :-
  gender(M1,male),
  gender(M2,male),
  gender(W,female),
  not (M1 == M2),
  home(M1,P1),
  home(W,P1),
```

```
  home(M2,P2),
  not (P1 == P2).

sit_obj(M1,
  (current_place(W,P2), believes(M1,loves(W,M2))),
  (current_place(W,P1), held_by(W,M1))) :-
  gender(M1,male),
  gender(M2,male),
  gender(W,female),
  not (M1 == M2),
  home(M1,P1),
  home(W,P1),
  home(M2,P2),
  not (P1 == P2).


% beliefs

believes(C,F,_) :-
  believes(C,F).
believes(C,F,S) :-
  belief(C,F,S),
  not added_belief(C,F).

added_belief(C,F) :-
  (F = (not F1),Fe = F1,!;
   Fe = F),
  (believes(C,Fe);
   believes(C,not Fe)).

belief(M1,loves(W,M2),S) :-
  gender(M1,male),
  gender(M2,male),
  not (M1 == M2),
  gender(W,female),
  home(M2,P2),
  (once((current_place(W,P2),
         current_place(M2,P2),
         not current_place(M1,P2),
         not held_by(W,M2)),S),!;
  holds(held_by(W,M1),S)).

belief(M1,not loves(W,M2),S) :-
  gender(M1,male),
  gender(M2,male),
  not (M1 == M2),
  gender(W,female),
  home(M2,P2),
  once((current_place(W,P2),held_by(W,M2)),S),
  not holds(held_by(W,M1),S).

beliefs :-
  log(L),
  forall(believes(A,F,L),describe(believes(A,F,L))).

template(believes(A,not F,S),[A,' does not believe that '|TF]) :-
  template(F,TF).
```

```
template(believes(A,F,S),[A,' believes that '|TF]) :-
  not (F == (not _)),
  template(F,TF).

user_controlled(believes(_,_)).

doubts(M,loves(W,M)) :-
  doubts(M,loves(W,M),start).

doubts(M,loves(W,M),S) :-
  gender(M,male),
  gender(W,female),
  not believes(M,loves(W,M),S),
  not believes(M,not loves(W,M),S).


% emotional conditions

emotional_condition :-
  log(L),
  forall(character(C),
    (once(emotional_condition(C,S,L)),
     describe(emotional_condition(C,S,L)))).

emotional_condition(C,S) :-
  emotional_condition(C,S,start).

emotional_condition(C1,absolutely_happy,S) :-
  gender(C1,female),
  emotional_condition(C1,happy,S),
  not (state(Si,S), holds(held_by(C1,_),Si)).

emotional_condition(C1,happy,S) :-
  character(C1),
  loves(C1,C2),
  not (character(C3),
       not (C3==C2),
       holds(held_by(C1,C3),S)),
  once((holds(current_place(C1,P),S),
        holds(current_place(C2,P),S))).

emotional_condition(C,bored,S) :-
  character(C),
  not emotional_condition(C,happy,S).

template(emotional_condition(C,S,P),[C,' is ',S]).


% INITIAL STATE

% general settings

state_rep([]).
op_level(0).
log(start).
```

```
% factual database - non-varying

place(forest).
place(palace).
place('Lanka').
character('Sita').
gender('Sita',female).
home('Sita',palace).
character('Rama').
gender('Rama',male).
home('Rama',palace).
character('Ravana').
gender('Ravana',male).
home('Ravana','Lanka').

% factual database - possibly varying

current_place('Sita',forest).
current_place('Rama',palace).
current_place('Ravana','Lanka').
loves('Rama','Sita').
loves('Ravana','Sita').
loves('Sita','Rama').
```

Appendix III - example pre-defined plots and plot libraries

```
% example of pre-defined plot

plot(fake_abduct(V,P)).

fake_abduct(V,W,P) :-
  P := [[f1:ride(V,P1,P2),f2:entreat(V,W),
         f3:seize(V,W),f4:carry(V,W,P1)],
        [f1-f2,f1-f3,f2-f4]].

% examples of Plot Libraries

lib(lib_1).
lib(lib_2).
lib(lib_3).

lib_1(
 [ [[f1:abduct('Meleagant','Guinevere'),
     f2:rescue('Lancelot','Guinevere')],
    [f1-f2]],
   [[f1:elope('Naoise','Deirdre'),
     f2:capture('Conchobar','Deirdre')],
    [f1-f2]],
   [[f1:abduct('Simbionese Liberation Army','Patricia Hearst'),
     f2:capture('San Francisco Police','Patricia Hearst')],
    [f1-f2]],
   [[f1:elope('Paris','Helen'),
     f2:rescue('Menelaus','Helen')],
    [f1-f2]] ]).
```

```
lib_2(
 [ [[f1:ride('Meleagant','Gore',forest),
     f2:seize('Meleagant','Guinevere'),
    f3:carry('Meleagant','Guinevere','Gore'),
    f4:ride('Lancelot','Camelot','Gore'),
    f5:defeat('Lancelot','Meleagant'),
    f6:entreat('Lancelot','Guinevere'),
    f7:carry('Lancelot','Guinevere','Camelot')],
    [f1-f2,f2-f3,f3-f4,f4-f5,f5-f6,f6-f7]],
   [[f1:ride('Tristan',forest,garden),
     f2:entreat('Tristan','Isolde'),
    f3:carry('Tristan','Isolde',forest),
    f4:ride('Mark','Cornwall',forest),
    f5:defeat('Mark','Tristan'),
    f6:seize('Mark','Isolde'),
    f7:carry('Mark','Isolde','Cornwall')],
    [f1-f2,f2-f3,f3-f4,f4-f5,f5-f6,f6-f7]] ]).

lib_3(L) :-
  L :=
   [sel []/(not loves(P,V)) @
    (ride(V,P1,P2) *
    (seize(V,P) + entreat(V,P)) *
    carry(V,P,P1))].
```