PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 07/11

Information-gathering Events in Story Plots

Fabio A. Guilherme da Silva
Antonio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 07/11 ISSN 0103-9741
Editor: Prof. Carlos José Pereira de Lucena August, 2011

Information-gathering Events in Story Plots

Fabio A. Guilherme da Silva
Antonio L. Furtado

fabio.guilherme@gmail.com, furtado@inf.puc-rio.br

Abstract: Story plots must contain, besides physical action events, a minimal set of information-gathering events, whereby the various characters can form their beliefs on the facts of the mini-world in which the narrative takes place. Three kinds of such events will be considered here, involving, respectively, inter-character communication, perception and reasoning. Multiple discordant beliefs about the same fact are allowed, making necessary the introduction of higher-level facilities to rank them and to exclude those that violate certain constraints. Since the proposed package was designed to run in a plan-based context, other higher-level facilities are also available for pattern-matching against typical-plan libraries or previously composed plots. A prototype logic programming implementation is fully operational. A simple example is used throughout the presentation.

Keywords: Plot Composition, Communicative Acts, Perception, Deduction, Abduction, Plan Recognition, Plan Generation, Logic Programming.

Resumo: Enredos de estórias devem conter, além de eventos de ações físicas, um conjunto mínimo de eventos de obtenção de informação, pelos quais os vários personagens possam formar suas crenças sobre os fatos do mini-mundo em que a narrativa tem lugar. Três espécies de tais eventos serão consideradas aqui, envolvendo, respectivamente, comunicação entre os personagens, percepção e raciocínio. Múltiplas crenças discordantes sobre o mesmo fato são permitidas, tornando necessária a introdução de facilidades de nível mais alto para graduá-las e para excluir as que violem certas restrições. Uma vez que o pacote proposto foi projetado para rodar em um contexto baseado em planos, outras facilidades de alto nível estão também disponíveis para casamento de padrões contra bibliotecas de planos típicos ou enredos previamente compostos. Um protótipo implementado em linguagem de programação em lógica está em pleno funcionamento. Um exemplo simples é usado ao longo da apresentação.

Palavras-chave: Composição de Enredos, Atos de Comunicação, Percepção, Dedução, Abdução, Reconhecimento de Planos, Geração de Planos, 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

Story plots typically include action events, but another class of events is also needed for the sake of realism: the *information-gathering* events, which enable the various characters to mentally apprehend the state of the world. Without such events, one would have to assume that the characters are omniscient, i.e. that they are aware of all facts that currently hold and of how they change as a consequence of the action events.

Here we shall recognize a sharp distinction between the facts themselves and the sets of *beliefs* of each character about the facts that hold at the current state of the world, which constitute, so to speak, their respective *internal states*. Beliefs can be right or wrong, depending on their corresponding or not to the actual facts. Moreover, we have taken the option that acquiring a belief does not cancel a previous belief. As a consequence, we allow a character to simultaneously entertain more than one belief with respect to the same fact, possibly with a different degree of confidence which depends on the provenance of the beliefs.

We shall consider three types of information-gathering events, each type associated with a set of operations:

- Communication events - operations: `tell`, `ask`, `agree`.
- Perception events - operations: `sense`, `watch`.
- Reasoning events - operations: `infer`, `suppose`.

All operations refer to beliefs on facts, except `watch`, whose object is some action event witnessed by a character. The operations are defined in terms of their pre-conditions and post-conditions [Fikes and Nilsson]. The pre-conditions are logical expressions commonly involving affirmed or negated facts and beliefs, whereas post-conditions denote the effect of the operation in terms of beliefs that are added or deleted to/from the current internal states of the characters involved.

However the specification of the operations is deliberately kept at a minimum, to be complemented, both with respect to pre-conditions and post-conditions, by separate *conditioners*, that express the peculiarities of the different characters participating in the stories.

As we have been doing in the course of our **Logtell** [Camanho et al.] storytelling project, a *plan-generation* algorithm is employed to compose plots as sequences of events, in view of the indicated goals. Since the present work focuses on the construction of an information-gathering package, to be later integrated to the design of full-fledged narrative genres, a single action event will be mentioned here to meet a requirement imposed by the information-gathering events. This single action event, associated with the `go` operation, consists of the displacement of a character from a place to another, which is needed because presential verbal interaction is the only form of communication that we currently cover.

All features discussed were implemented in a logic-programming prototype, and a simple running example is used throughout the paper as illustration. Section 2 briefly explains how the example was formulated so as to run in a plan-based context. Section 3 describes the three types of information-gathering events. Section 4 adds some higher-level facilities, which help to analyze the resulting beliefs and to make comparisons by means of *plan-recognition*. Section 5 contains concluding remarks.

2. Example in a plan-based context

2.1. Conceptual specification

Our conceptual design method involves three schemas: static, dynamic and behavioural.

The *static schema* specifies, in terms of the *Entity-Relationship* model [Batini et al.], the entity classes, attributes and binary-relationships. The information-gathering package minimally requires the clauses below (see Appendix, **PART 1**):

```
entity(person,name).
attribute(person,gender).
relationship(current_place,[person,Place]) :-
  taken_as_place(Place).
attribute(person,believes).
attribute(person,told).
attribute(person,sensed).
attribute(person,watched).
attribute(person,inferred).
attribute(person,supposed).
attribute(person,asked).
relationship(trusts,[person,person]).
```

Notice that the specification of the `current_place` relationship associates the entity `person` with a still undetermined entity, represented by the variable `PLACE`. When the package is put together with a running application, other clauses can be added (Appendix, **PART 7**), in particular to choose what sort of location will correspond to `PLACE`:

```
entity(country,country_name).
entity(city,city_name).
attribute(person,hair_colour).
attribute(person,daltonic).
relationship(born,[person,country]).
relationship(home,[person,country]).
relationship(citizen,[person,country]).

taken_as_place(city).
```

The *dynamic schema* defines a fixed repertoire of operations for consistently performing the state changes corresponding to the events that can happen in the mini-world of the application. The *STRIPS* [Fikes and 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. The operations that constitute the core of the information-gathering package will be described in section 3.

Currently our *behavioural schema* specifications mainly consist of goal-inference (a.k.a. situation-objective) rules. Since our present running example does not employ such rules, we shall not discuss them here (cf. [Camanho et al; Ciarlini et al]).

2.2. Initial state declarations

To run an application, it is necessary to populate the initial database state with ground clauses denoting valid instances of the specified static schema (for their formal representation in our example, see the Appendix, **PART 11**).

Informally speaking, the mini-world of our example comprises four characters, John, Peter, Mary and Laura, three countries, UK, USA and Canada, and two cities, both in the UK, London and Manchester.

The recorded information does not provide a uniform coverage. It registers where Mary, Peter and Laura were born but does not indicate John's birth-place. About Mary it adds that her domicile (home) is also in the UK and that she has red hair, whereas Laura ─ who, in spite of having been born in the USA, is a Canadian citizen ─ is blond. Peter is said to be daltonic. John, Peter and Mary are currently in London, and Laura in Manchester.

Contrary to the other characters, whose beliefs are initially confined to their explicitly recorded properties, John is aware of all registered facts.

.

2.3. 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 this is not the case, 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`.

In view of the needs of the information-gathering package, we specified `added(pre_state(O,S),O)` as one more effect of every operation `O`, which allows to capture in `S` the *prefix* of operation `Op`, i.e. the entire plan sequence, starting at the initial state, to which `Op` will be appended. Indeed, sequence `S` supplies a convenient operational denotation of the state immediately before the event denoted by `Op`, to which we may, in particular, apply the `holds` predicate to find out who was present at some place associated with the occurrence of the event (cf. the `precond` clause for operation `watch` in the Appendix, **PART 2**).

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

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 can be 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 `tell(A,B,F)` (in the Appendix, **PART 2**), where character `A` tells something to character `B`. We require that the two characters should be together at the same place, and, accordingly, the `Pr` argument shows two terms containing the same variable `L` to express this location requirement, but the term for `B` is barred: `/current_place(B, L)`, which does not happen in `A`'s case. The difference has an intuitive justification: character `A`, who is the agent of the operation, has to either already be present or to go to the place `L` where `B` is, but the latter would just happen to be there for some other reason.

The proper treatment of (a) and (b) is somewhat tricky, because of the backward chaining strategy of the planning algorithm. 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. Delayed evaluation is also needed, as one would expect, for instantiating the `pre_state` predicate mentioned before.

Only for the sake of completion, since this feature is not used in the package, we still have to mention that, for 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.

Once generated, a plan can be processed via the execute command, thus effecting the desired state transition, i.e. adding and/or deleting facts to/from the current database state. As a side effect, the `log(S)` clause (initially set as `log(start)`) is updated by appending to `S` the plan executed. At any time, the entire story thus far composed can be narrated, in pseudo-natural language, by simply entering "`:- log.`".

To finish this partial review of the plan features, we remark that the planning algorithm `plans(G,P)` can be called in more than one way. More often `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 executable in view of the initial state and of the interplay of pre- and post-conditions, and, if so, assign its net effects (a conjunction of `F` and `not F` terms) to `G`. This second kind of usage will be illustrated in example 11 of section 4.3.

2.4. Templates for pseudo-natural language generation

Both for facts and events, we resort to *templates* for description and narration in pseudo-natural language (the pertinent clauses are given in the Appendix, **PARTS 3, 4, 8**, as well as in **PART 2** for each operation).

The template device allows, to begin with, to list all properties registered in the initial database state (or in the current state reached by executing a plan), via the facts predicate:

```
:- facts.

    John is a person
    Mary is a person
    Peter is a person
    Laura is a person
    UK is a country
    USA is a country
    Canada is a country
    London is a city
    Manchester is a city
    Mary is female
    Laura is female
    John is male
    Peter is male
    Mary has red hair
    Laura has blond hair
    Peter is daltonic
    John is now in London
    Peter is now in London
    Mary is now in London
    Laura is now in Manchester
    Mary was born in UK
    Peter was born in UK
    Laura was born in USA
    Mary lives in UK
    Laura is a citizen of Canada
```

The templates for operations and facts are combined in a way that favours a fairly readable style. Consider, as an example, operation `tell(A,B,F)` as specified in the Appendix, **PART 2**. Suppose fact `F` corresponds to a property of character `C`. Concerning the identity of the three characters, we can distinguish three situations:

- They are all distinct - `tell('John','Peter',hair_colour('Mary',red))`
- C is the same as A - `tell('Mary','Peter',hair_clour('Mary',red))`
- C is the same as B - `tell('John','Mary',hair_colour('Mary',red))`

The `def_template` algorithm (Appendix, **PART 4**) that drives the application of the templates produces for the above events:

```
John tells Peter: "- Mary has red hair".
Mary tells Peter: "- My hair is red".
John tells Mary: "- Your hair is red".
```

The algorithm duly uses gender information, rendering for example `infer('Mary',citizen('Mary','UK'))` as:

```
Mary infers that she is a citizen of UK.
```

Negative facts in the several operations, and the occurrence of variables in the `ask` operation, are treated as expected by the algorithm. So, again for the `hair_colour` property, one will have, respectively:

```
tell('John','Laura',not hair_colour('Laura',red))
   John tells Laura: "- Your hair is not red".
ask('John','Laura',hair_colour('Laura',X))
   John asks Laura: "- What is the colour of your hair?".
ask('John','Laura',hair_colour(X,red)))
   John asks Laura: "- Who has red hair?".
```

Analogous templates are provided for rendering in pseudo-natural language the information-package facts, such as `told`, `asked`, `sensed`, `watched`, `inferred`, `supposed`, `believes` and `trusts`. They can be listed at any current state by entering "`:- info_facts.`".

3. The information gathering events

The complete specification of the operations associated with the information-gathering events (and of the auxiliary `go` operation) is given in the Appendix, **PART 2**.

3.1. Communication events

In the computer science community, communication between characters immediately brings to mind the communication processes executed by software agents in multi-agent environments. In particular, the Agent Communication Language proposed by the Foundation for Intelligent Physical Agents consists of formally defined operations similarly defined by their pre-conditions and post-conditions [FIPA]. Software agents differ from fictional characters (and, ironically, from human beings in general) in that they are supposed to only transmit information on which they believe, to agents that still lack such information and need it in order to play their role in the execution of some practical service.

In contrast, certain characters are prone to lie, either for their benefit or even out of habit. In general they may ignore the conversational maxims prescribed by philosophers of language, such as [Grice]. The bare specification of our `tell(A,B,F)` operation does not even require that `A` has any notion of the fact `F` to be transmitted to `B`. It is enough that both characters are at the same local `L`; if they are not, a `current_place(A,L)` sub-goal is recursively activated, which may cause the displacement of the teller (character `A`) to `L`, where `B` currently is (note the "/" sign before `current_place(B,L)`, to indicate that B would not be expected to move). And the only necessary effect of the operation is merely

that F is told by A to B. Whether or not B will believe in F will depend on the execution of the agree operation, which in turn depends on whether or not B trusts in A.

The ask operation is similarly defined, and its effect is just that A has asked F from B, who may or may not respond. The fundamental character-dependent conditioners (all spelled out in the Appendix, **PART 9**) are established, respectively, by separate will_tell and will_ask clauses.

Example 1: Mary is willing to ask John about his current whereabouts. She asks, he replies and, since she trusts him, adopts the belief that he is in London.

```
:- iter(1, (believes('Mary',current_place('John',L)))).

Goal: believes(Mary, current_place(John, A))

Mary asks John: "- Where are you?". John tells Mary: "- I am now in
London". Mary agrees with John.
```

3.2. Perception events

Perception is the faculty whereby people keep contact with the world through their five senses (sight, hearing, touch, smell and taste). At the present stage of our work we do not make such distinctions, and merely consider a generic sense operation to apprehend any sort of fact, with a variant version that makes provision for defective sensing. For correct sensing of a positive or negative fact F, F must be successfully tested. Distorted sensing is accompanied by a side-remark on the true fact. In any case, besides the sensed clause (analogous to the told and asked clauses of the communication operations) a belief clause is immediately added, since direct perception does not depend on a third party who might not be trusted.

The watch(W,O) operation was harder to implement, requiring the inclusion of the already mentioned pre_state(O,S) clause as an extra feature of the plan-generator, where O is the operation witnessed by W, and S denotes the state previous to the application of O (i.e. the sub-sequence of the generated plan that precedes O). It becomes possible then to check the location of character W at the time when O happens.

As before, the definitions are left to be completed by conditioners, respectively sense-rule and watch_rule clauses. For sense, it is required that, to ascertain a positive or negative fact F involving an entity instance E currently at place L, a character W must be at L, either originally or as the result of pursuing current_place(W,L) as a sub-goal. Note that this requirement holds even if F is the negative fact not current_place(E,L), since W must be at L to sense that E is not there.

For watch, normally applicable to action events only, the current_place requirements depend on what is being watched, which justifies their being left to the special watch_rule clauses, to which the current_place(W,L) information, checked as described before, is passed. For instance, go(A,L1,L2) can be watched partly by persons then at L1 (origin) and partly by those present at L2 (destination). Naturally the agent (character A) is able to watch the action in its entirety.

Example 2: Peter obtains three different indications concerning the colour of Mary's hair. Only the first, supplied by trustworthy John, was correct. Laura was lying, and Peter himself, being daltonic, failed to perceive the true colour.

```
:- iter(3,believes('Peter',hair_colour('Mary',A))).

Goal: believes(Peter, hair_colour(Mary, A))

John tells Peter: "- Mary has red hair". Peter agrees with John.

Laura goes from Manchester to London. Laura tells Peter: "- Mary has
blond hair". Peter agrees with Laura.

Peter wrongly senses that Mary has green hair -- in fact Mary has red
hair.
```

Example 3: The daltonic Peter tries to convince himself that neither Mary nor Laura have red hair. His eyes seem to confirm that in both cases. He is wrong with respect to Mary, but quite right with respect to Laura, since he would not have trouble distinguishing red from the yellow hue of Laura's hair.

```
:- iter(1,(sensed('Peter',not hair_colour('Mary',red)), sensed('Peter',
not hair_colour('Laura',red)))).

Goal: sensed(Peter, not hair_colour(Mary, red)), sensed(Peter, not
hair_colour(Laura, red))

Peter wrongly senses that Mary has not red hair. Peter goes from London
to Manchester. Peter senses that Laura has not red hair
```

Example 4: John travels from London to Manchester, and Peter, being in London, watches the event and reports it to Mary.

```
:- iter(1, told('Peter', ['Mary', watched('Peter', go('John', 'London',
'Manchester'))])).

Goal: told(Peter, [Mary, watched(Peter, go(John, London, Manchester))])

John goes from London to Manchester. Peter watches the event: 'John goes
from London to Manchester'. Peter tells Mary: "- I watched the event:
'John goes from London to Manchester'".
```

3.3. Reasoning events

Deduction, induction and abduction are complementary reasoning strategies. For deduction, if there is a rule $A \rightarrow B$ and the antecedent A is known to hold, it is legitimate to *infer* that the consequent B holds. In the case of induction (fundamental to the natural sciences), the systematic occurrence of B whenever A occurs may justify the adoption of rule $A \rightarrow B$.

Abduction (cf. [Peirce], for example) is a non-guaranteed but nevertheless most useful resource in many uncertain situations: given the rule $A \rightarrow B$, and knowing that B holds,

one may *suppose* that *A* also holds. This is a type of reasoning habitually performed by medical doctors, who try to diagnose an illness in view of observed symptoms. The trouble is, of course, that it is often the case that more than one illness may provoke the same symptom — in other words: there may exist other applicable rules $A^1 \rightarrow B$, $A^2 \rightarrow B$, ..., $A^n \rightarrow B$, suggesting different justifications for the occurrence of *B*. In abduction, wherein the implication arrow is followed backward, one is led to formulate hypotheses rather than the firm conclusions issuing from deduction over deterministic rules.

Our `infer` and `suppose` operations utilize, respectively, deduction and abduction. The conditioners for both can be the same rules of inference (`inf_rules`) to be traversed forward in the former case or backward in the latter. The reader will notice that, in the case of `infer`, a rule `P=>F` accepted by character `A`, the antecedent `P` furnishes the beliefs to be tested as pre-condition, whereas `A`'s belief in `F` will be acquired as the `added` effect (another addition being an `inferred` clause) upon a successful evaluation of `P`. Conversely, in the case of `suppose`, the belief on the consequent will motivate the addition of a belief in some fact present in the logical expression of the antecedent.

We must stress that the inference rules adopted by one or more characters in a given story do not have to be scientifically established rules. Often originating from popular traditions, they may lead to far-fetched or even absurd beliefs. Informally, our rules are:

a. if person A was born in a country B, and A's domicile is also in B, then A is a citizen of B.
b. if person A is daltonic, and says that the colour of B's hair is C1, but a daltonic when looking at an object coloured C2 would mistakenly perceive that colour as C1, then the colour of B's hair is C2.
c. if person A was observed departing from location L to some other location, then A is no longer at L.
d. if person A was observed arriving at location L, coming from some other location, then A is currently at L.

Rules **c** and **d** are trivial, and we only included then as examples of a more general case: watching an event from an appropriate place allows the observer to conclude that the direct and indirect effects of the event should hold. More interesting (but not present in our reduced context) is what one may conclude from watching a kidnap scene: not only that the victim has lost all freedom of movement, but that the author of the villainy would take the victim to his own dwelling. Rule **a** does not really cover the legal citizenship requirements prevailing in most countries; indeed legal argument goes far beyond the scope of classical syllogism, as expounded for instance in [Toulmin]. Rule **b** represents a naive understanding (taking red for green and vice-versa) of one variety of colour blindness (cf. [Dalton] for an early study, noting that the word "daltonism" derived from that author's name).

Example 5: John decides to leave London and go somewhere else. Manchester is chosen (by the plan-generator) as destination. Peter, who is in London, senses that John is no longer there, whereas Laura, living in Manchester, now senses John's presence. A trivial inference from watching John's departure or arrival provides a second way to obtain the same information.

```
:- iter(3,
     (not current_place('John','London'),
      believes('Peter',not current_place('John','London')),
      believes('Laura',current_place('John',L)))).

Goal: not current_place(John, London), believes(Peter, not
current_place(John, London)), believes(Laura, current_place(John, A))

John goes from London to Manchester. Peter senses that John is not in
London now. Laura senses that John is now in Manchester.

John goes from London to Manchester. Peter senses that John is not in
London now. Laura watches the event: 'John goes from London to
Manchester'. Laura infers that John is now in Manchester.

John goes from London to Manchester. Peter watches the event: 'John goes
from London to Manchester'. Peter infers that John is not in London now.
Laura senses that John is now in Manchester.
```

Example 6: Peter wonders about Mary's nationality. Since John knows in what country she was born, which is equally her domicile, he infers that she is a citizen of that country and correctly informs Peter. Laura's indication is false — and also not valid, since citizenship refers to countries, not to towns. The third sequence is more curious, and we must confess that, though it looks rather obvious, we were not expecting it: John passes the two antecedents to Peter, thus enabling him to deduce the consequent by himself.

```
:- iter(3,believes('Peter',citizen('Mary',L))).

Goal: believes(Peter, citizen(Mary, A))

John infers that Mary is a citizen of UK. John tells Peter: "- Mary is a
citizen of UK". Peter agrees with John.

Laura goes from Manchester to London. Laura tells Peter: "- Mary is a
citizen of London". Peter agrees with Laura.

John tells Peter: "- Mary was born in UK". Peter agrees with John. John
tells Peter: "- Mary lives in UK". Peter agrees with John. Peter infers
that Mary is a citizen of UK.
```

Example 7: Conversely, the question here concerns Laura's domicile, which is one of the antecedents of the inference rule involved in example 6. Knowing that Laura is a Canadian citizen, John supposes, by traversing the inference rule in the opposite direction, that Canada is her home, and passes his belief to Peter. The belief may or may not be correct, there being no contrary evidence at least. But notice that, if the question concerned Laura's birth-place, the definitely incorrect supposition that this was also Canada would result, which shows how careful one should be when resorting to abduction — a most useful way to formulate hypotheses, to be later confirmed or not through additional evidence.

```
:- iter(1,believes('Peter',home('Laura',L))).

Goal: believes(Peter, home(Laura, A))
```

```
John supposes that Laura lives in Canada. John tells Peter: "- Laura
lives in Canada". Peter agrees with John.
```

Example 8: As a goal that, so we thought, should fail, we enquired how could red-haired Mary come to believe that her hair was not red. To our surprise, the planner found a solution using inference in a most devious way: John gives Peter the correct information, which he faithfully transmits to Mary. However, having first noticed that Peter is daltonic, Mary is led to apply our (admittedly naive) inference rule dealing with red-green colour blindness.

```
:- iter(1,(believes('Mary',hair_colour('Mary',C)),not (C = red))).

Goal: believes(Mary, hair_colour(Mary, A)), not A=red

Mary senses that Peter is daltonic. John tells Peter: "- Mary has red
hair". Peter agrees with John. Mary asks Peter: "- What is the colour of
my hair?". Peter tells Mary: "- Your hair is red". Mary infers that she
has green hair.
```

4. Higher-level facilities

In the current prototype, the predicates implementing the higher-level facilities (given in the Appendix, **PART 5**) do not correspond to events to be inserted by the planner in a story plot. They are, so to speak, external to the narrative, to be applied by the user currently in control of the story composition, as an instrument of analysis. Future work may promote their inclusion in the repertoire of events, especially in the context of detective stories, where the critical examination of past facts and events plays a fundamental role (more on that in section 5). To run the higher-level facilities with a specific application, a number of clauses must be specified (for those used in our example, see the Appendix, **PART 10**).

4.1. Surveying and ranking multiple beliefs

Since multiple beliefs about the same fact are allowed, one needs a device to rank them on the basis of their provenance. The survey predicate collects all beliefs of a character about an indicated fact, together with their provenance (operation whereby they were acquired) and puts them in decreasing order with respect to the appropriate weights. These must have been declared by a conditioner such as `weights('Peter', hair_colour(X,Y), [1:sensed('Peter',_),2:told('John',['Peter',_])])`, which is utilized in example 9 below.

Example 9: Peter collects all possible inputs on the colour of Mary's hair (as in example 2). He then ranks the results, according to his pre-defined list of weights based on provenance. As far as hair-colour is concerned, what he hears from John is ranked (weight 2) above the evidence of his own defective eyesight (weight 1). He trusts Laura, but never thought of assigning a weight to her opinion (weight 0 is the default).

```
:- survey('Peter',hair_colour('Mary',C),F:V),
  rank('Peter',F:V,Vr),
  nl,varnames(F),
  write('Surveying: '),write(F),nl,nl,
  forall(member(Pr:_,V),(write(Pr),nl)),nl,
  write('Ranking the results: '),nl,nl,
  forall(member(P,Vr),(write(P),nl)), nl, nl, !.


Surveying: hair_colour(Mary, A)

sensed(Peter, hair_colour(Mary, green))
told(John, [Peter, hair_colour(Mary, red)])
told(Laura, [Peter, hair_colour(Mary, blond)])

Ranking the results:

2:hair_colour(Mary, red)
1:hair_colour(Mary, green)
0:hair_colour(Mary, blond)
```

The `survey_v` predicate (also in the Appendix, **PART 5**), when collecting the various inputs, rejects those that cause the activation of any `violate_rule`. Such rules play a central role in the higher-level facility described in the next section.

4.2. Validating a belief

Certain beliefs may not make sense in that they violate some natural law, or legal norm or even some convention of the chosen story genre. An elementary kind of violation refers to the schema definition itself: e.g. instances of a relationship are not acceptable if not declared between instances of the entity classes over which the relationship was defined.

Also, similarly to naive inferences, a `violate_rule` established for a story genre may not reflect precisely the real world.

The `violations` predicate, illustrated below, checks a given expression in view of the established `violate_rules`. If a rule is violated more than once, the rule identifier will be repeated an equal number of times in the resulting list.

In our example, both `violate_rules` are about the `citizen` relationship. According to rule r1, any instance thereof is considered not valid if the first parameter is not a `person` or the second is not a `country`, whereas rule r2 excludes the possibility of plural citizenship (although, sometimes contrary to official legislation, such cases are often encountered in practice).

Example 10: When checking expression `A` below, rule r1 is activated twice: London is not a `country` and Mickey is not a `person`. A violation related to rule r2 is also detected, since there should be no more than one clause declaring Mary's citizenship.

```
:- A = (citizen('Mary','UK'), citizen('Mary','London'),
        citizen('Mickey','USA')),
  violations(A,Vs),
```

```
  nl,write('Clause: '),write(A),nl,
  write('violates rules: '),write(Vs),nl,nl,nl.

Clause: citizen(Mary, UK), citizen(Mary, London), citizen(Mickey, USA)
violates rules: [r1, r1, r2]
```

4.3. Recognizing a library plan from events observed

Typical plans can be extracted from previously existing plots, and, after their parameters are consistently replaced by variables, be stored under this plan-pattern form in a library, for future reference. Our tiny example library comprises two short plans:

```
lib([
  start=>go(A,L1,L2)=>go(B,L1,L2)=>sense(A,current_place(B,L2)),
  start=>go(A,L1,L2)=>go(A,L2,L1)
 ]).
```

In the former, two characters `A` and `B` follow the same itinerary and, next, `A` senses that they are now together at place `L2`. The latter just shows `A` leaving from and returning to the same place.

One of the uses of a library is to match one or more observed events against each plan-pattern. If all the observations supplied, ideally in a small number, unify with plan events that must lie in the same sequence but do not have to be contiguous, one gains the following complementary benefits:

a) anticipating what the characters are trying to achieve in the long run.
b) extending the few events to a more comprehensive plot, consisting of the matching plan pattern, with some (or all) variables instantiated due to the unification.

To obtain further intuitive understanding of what (a) means, take the commonplace example of a person being observed to hail a taxi and go to an airport. These observations would match a plan-pattern with events such as buying an air ticket, hailing a taxi, loading a number of bags on the taxi, going to the airport, etc., etc., checking-in, boarding the plane, etc. But they might also match a similar plan in which the person would be going to the airport not to embark but to meet another person in an arriving flight. The fact that the same observations can match alternative plans shows that recognition can, in general, be hypothetical.

On the other hand, no matter if with just one or with several matching alternatives, a prospective author would have, in view of (b), two possible plots obtained by extending an initial fragmentary sketch. So, curiously, both plan-generation (as illustrated in the previous sections) and plan-recognition provide useful story composition strategies. Plan-recognition brings to mind the notion of reuse, and in the literary domain is in consonance with the remark in [Barthes] that "any text is a new tissue of past citations".

Example 11: Two observed `go(X,Y,Z)` events are matched against the given library of typical plans. The first event is fully instantiated, whereas the second seems to result from a vague observation: the only clue is that the agent was either John himself or a woman. With

the first option, the library plan wherein the same character travels forward and then backward is recognized; with the second, the recognized plan is that two different characters embark on the same trip, and the former notices the presence of the latter when they have both reached their destination. Calling the plan-generator to validate the plans has the effect of restricting the choice of the female character to Mary, who happens to be initially in London as required.

```
:- Obs = [go('John','London','Manchester'),go(A,L1,L2)],
  Assuming = (A = 'John'; gender(A,female)),
  copy((Obs,A),(Obs1,A1)),varnames(Obs1),
  nl,write('Observed: '),write(Obs1),nl,
  write('assuming that '),write(A1),write(' was either John himself'),nl,
  write('or some person of the female gender'),
  nl,nl,nl,
  write('Library plans recognized - and executable:'),nl,!,
  lib(L),
  forall((Assuming,recognize_lib(Obs,P,L)),
    (plans(_,P),varnames(P),narrate(P); not plans(_,P))),
  nl, nl.

Observed: [go(John, London, Manchester), go(A, B, C)]
assuming that A was either John himself
or some person of the female gender

Library plans recognized - and executable:

John goes from London to Manchester. John goes from Manchester to London.

John goes from London to Manchester. Mary goes from London to Manchester.
John senses that Mary is now in Manchester.
```

4.4. Recognizing a pattern in a generated plan

Pattern-matching can also take an opposite direction, working on an existing plot and checking whether it contains some not necessarily contiguous subsequence to which a pattern may be matched successfully. Both verifying that the match succeeds and, if so, extracting the matching subsequence are important contributions towards the analysis of plots.

Example 12: A pattern expressing a going and returning trip performed by some character is matched against an existing plot, which, among its events, contains an instance of the pattern ─ which is duly found and displayed.

```
:- Evs = [go(A,L1,L2),go(A,L2,L1)], copy(Evs,Evs1),
  H = start=>go('John', 'London', 'Manchester')=>go('Laura',
   'Manchester','London')=>go('John', 'Manchester', 'London'),
  check_obs(Evs,H),
  nl,write('Applying the pattern: '),varnames(Evs1),write(Evs1),nl,
  write('to the given sequence: '),write(H), nl,
  write('one finds: '),write(Evs),
  nl,nl,nl,!.
```

```
Applying the pattern: [go(A, B, C), go(A, C, B)]
to the given sequence: start=>go(John, London, Manchester)=>go(Laura,
  Manchester, London)=>go(John, Manchester, London)
one finds: [go(John, London, Manchester), go(John, Manchester, London)]
```

5. Concluding remarks

Besides the simple-minded example used as illustration, we have already applied some of the information-gathering events to enhance the *Swords-and-Dragons* genre that runs in **Logtell**, by dropping the unrealistic omniscience assumption. Now a damsel sees the villainous Draco kidnapping princess Marian, and runs to tell sir Brian about the mischief; the knight immediately infers that a kidnapped victim should have been carried to the villain's dwelling, whereto he promptly rides to rescue his beloved.

But the availability of information-gathering events will, probably after further elaboration, open the way to more sophisticated genres. In particular, we have been examining the requirements of *detective stories*. It has been convincingly argued [Todorov] that such narratives actually contain two stories: the story of the crime and the story of the investigation. The first story, that of the crime, ends before the second begins; the characters of the second story, the story of the investigation, do not act, they learn ─ which is well within the scope of the package discussed here.

Future work should extend the repertoire of events to contemplate other speech-acts [Austin; Searle; Leech and Weisser], for example to allow a character C1 to solicit or to order another character C2 to execute an action of C1's interest, which C2, but not C1, is empowered and in a position to perform.

Moreover, in stories of even moderate complexity, behaviour should be characterized as a decision-making process affecting the participation of each character in every kind of event, involving physical action or the information-gathering activities of the present study ─ and this process hinges on both cognitive and emotional considerations [Brave and Nass; Loewenstein and Lerner; McCrae and Costa; Goldberg; O'Rorke and Ortony; Ortony], further influenced by the goals and plans of the other characters [Willensky]. We have done some initial work on drives, attitudes, emotions, and mutual interferences among agents [Barbosa et al.], but a full integration within the **Logtell** system still remains to be achieved.

References

Austin, J.L. (1962). *How to do things with words*. London: Oxford University Press.

Barbosa, S.D.J.; Furtado, A.L.; Casanova, M.A.C. (2010). "A Decision-making Process for Digital Storytelling". Proc. *IX Symposium on Computer Games and Digital Entertainment - Track: Computing*.

Batini, C.; Ceri, S.; Navathe, S. (1992). *Conceptual Design – an Entity-Relationship Approach*. Benjamin Cummings.

Barthes, R. (1981). "The Theory of the Text". In *Untying the Text: A Post-Structural Reader.* R. Young (ed.). Boston: Routledge, 31-47.

Brave, S.; Nass, C. (2008) "Emotion in Human-Computer Interaction". In: A. Sears & J.Jacko (eds.) *The Human-Computer Interaction Handbook*, pp. 77–92.

Camanho, M.M; Ciarlini, A.E.M.; Furtado, A.L; Pozzer, C.T.; Feijó, B. (2008). "Conciliating coherence and high responsiveness in interactive storytelling". Proc. of the *3rd International conference on Digital Interactive Media in Entertainment and Arts*, pp. 427-434.

Ciarlini, A.E.M; Barbosa, S.D.J.; Casanova, M.A; Furtado, A.L. (2008). "Event relations in plan-based plot composition". Proc. *VII Symposium on Computer Games and Digital Entertainment - Track: Computing*, pp. 31-40.

Dalton, J. (1798). "Extraordinary facts relating to the vision of colours: with observations". *Memoirs of the Literary and Philosophical Society of Manchester* **5**: 28–45.
FIPA Communicative Act Library Specification (2002). At http://www.fipa.org/specs/fipa00037/SC00037J.html.
Fikes, R.E.; Nilsson, N.J. (1971). "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence* , 2(3-4).
Goldberg, L.R. (1992). "The Development of Markers for the Big-Five Factor Structure". *Psychological Assessment*, v4, n1 pp. 26-42.
Grice,H.P. (1975). "Logic and conversation". In: P. Cole, J.L. Morgan (Eds.), *Syntax and Semantics, Speech Acts*, vol. 3, Academic Press, New York.
Leech, G. and Weisser, M. (2003). 'Generic speech act annotation for task-oriented dialogues', in D. Archer, P. Rayson, A. Wilson and T. McEnery (eds.) *Proceedings of the Corpus Linguistics 2003 conference*, University Centre for Computer Corpus Research on Language, Technical Papers 16.1, pp. 441-6.
Lloyd, W. 1987. *Foundations of Logic Programming*. Springer.
Loewenstein, G.; Lerner, J.S. (2003). "The role of affect in decision making". In *Handbook of Affective Sciences*. Davidson, R.J.; Scherer, K.R.; Goldsmith, H.H. (eds.). Oxford University Press, pp. 619-642.
McCrae, R.R.; Costa, P.T. (1987). "Validation of a five-factor model of personality across instruments and observers". *J. Pers. Soc. Psychol.*, 52, pp. 81-90.
O'Rorke, P.; Ortony, A. (1994). "Explaining Emotions". *Cognitive Science*, 18, 2, pp. 283-323.
Ortony, A. (2003). "On making believable emotional agents believable". In *Emotions in Humans and Artifacts*. Trappl, R.; Petta, P.; Payr, S. (eds). The MIT Press, pp. 189-211.
Peirce, C.S. (1931). *Collected Papers*. Harvard University Press, Cambridge, MA (excerpted in J. Buchler (Ed.), Philosophical Writings of Peirce, Dover, New York, 1955).
Searle, J.R. (1979). *Expression and Meaning*, Cambridge University Press, Cambridge.
Todorov, T. (1977). *The Poetics of Prose*. Cornell University Press.
Toulmin, S.E. (1958). *The Uses of Argument*. Cambridge, Cambridge University Press.
Willensky, R. (1983). *Planning and Understanding - a Computational Approach to Human Reasoning*. Addison-Wesley.

Appendix

```
/* INFORMATION GATHERING PACKAGE */

/* preparatory commands */

:- set_prolog_flag(verbose,silent).
:- style_check([-singleton,-discontiguous]).
:- set_prolog_flag(toplevel_print_options,[max_depth(50)]).

:- dynamic told/2, sensed/2, watched/2, inferred/2, supposed/2, asked/2,
state_rep_ini/1, ini/0 , believes/2 .
:- dynamic person/1, gender/2, current_place/2.
:- dynamic it_m/1 .


% including and preparing for the planning algorithm

:- op(900,fy,not).
:- op(650,yfx,=>).
:- op(500,fx,/).
:- dynamic '/'(/1 ).
:- dynamic log/1 .
log(start).
log :-
  once(narrate),nl.
:- include(warbeta_info).
added(X,Y) :- /added(X,Y).
deleted(X,Y) :- /deleted(X,Y).
added(pre_state(O,S),O).


/* PART 1 - general information-gathering properties */

entity(person,name).
attribute(person,gender).
relationship(current_place,[person,Place]) :-
  taken_as_place(Place).
attribute(person,believes).
attribute(person,told).
attribute(person,sensed).
attribute(person,watched).
attribute(person,inferred).
attribute(person,supposed).
attribute(person,asked).
relationship(trusts,[person,person]).


/* PART 2 - general purpose dynamic schema */

% communication operations

operation(tell(A,B,F)).
added(told(A,[B,F]),tell(A,B,F)).
precond(tell(A,B,F),P) :-
  will_tell(A,B,F,P1),
  appc((current_place(A,L),/current_place(B,L)),P1,P).
template(tell(A,B,F),[A,' tells ',B,': "- '|Ft]) :-
  def_template(F,A,B,Ft1),
  append(Ft1,['"'],Ft).
```

```
operation(ask(A,B,F)).
added(asked(A,[B,F]),ask(A,B,F)).
precond(ask(A,B,F),P) :-
  will_ask(A,B,F,P1),
  appc((current_place(A,L),/current_place(B,L)),P1,P).
template(ask(A,B,F),[A,' asks ',B,': "- '|Ft]) :-
  def_template(F,int,A,B,Ft1),
  append(Ft1,['?"'],Ft).

operation(agree(A,B,F)).
added(believes(A,F),agree(A,B,F)).
precond(agree(A,B,F),(told(B,[A,F]),trusts(A,B))).
template(agree(A,B,F),[A,' agrees with ',B]).

% perception operations

operation(sense(W,F)).
added(believes(W,F),sense(W,F)).
added(sensed(W,F),sense(W,F)).
precond(sense(W,F),P) :-
  sense_rule(W,F,P1),
  (not (F = (not current_place(_,_))),
   appc((/F,property(F,Pr,I),current_place(W,L),/current_place(I,L)),P1,P);
   F = (not current_place(I,L)),
   appc((/F,current_place(W,L)),P1,P)).
template(sense(A,F),[A,' senses that '|Ft]) :-
  def_template(F,A,nil,Ft).

operation(sense(W,F1,F2)).
added(believes(W,F2),sense(W,F1,F2)).
added(sensed(W,F2),sense(W,F1,F2)).
precond(sense(W,F1,F2),P) :-
  sense_rule(W,F1,F2,P1),
  appc((property(F1,Pr,I),current_place(W,L),/current_place(I,L)),P1,P).
template(sense(W,F,F),[W,' senses that '|Ft]) :-
  def_template(F,W,nil,Ft).
template(sense(W,F1,F2),[W,' wrongly senses that '|Ft]) :-
  def_template(F2,W,nil,Fta),
  def_template(F1,W,nil,Ftb),
  (F1 = (not _),!,Ft = Fta;
   append(Fta,[' -- in fact '|Ftb],Ft)).

operation(watch(W,O)).
added(watched(W,O),watch(W,O)).
precond(watch(W,O),(pre_state(O,S),T,holds(current_place(W,L),S))) :-
  watch_rule(W,O,R,L), (R = true, T = true; not (R = true), T = holds(R,S)).
template(watch(W,F),[W,' watches the event: '''|Ft]) :-
  template(F,Ft1),
  append(Ft1,['''],Ft).

% reasoning operations

operation(infer(A,F)).
added(believes(A,F),infer(A,F)).
added(inferred(A,F),infer(A,F)).
precond(infer(A,F),P) :- ded(A,F,P).
template(infer(A,F),[A,' infers that '|Ft]) :-
  def_template(F,A,nil,Ft).

ded(A,F,Fc) :-
  inf_rule(A,P=>F),
  conj_list(P,L),
  ded1(A,L,Lc),
```

```
  conj_list(Fc,Lc).

ded1(A,[],[]).
ded1(A,[X|R],[believes(A,X)|S]) :-
  property(X),!,
  ded1(A,R,S).
ded1(A,[X|R],[X|S]) :-
  ded1(A,R,S).

operation(suppose(A,F)).
added(believes(A,F),suppose(A,F)).
added(supposed(A,F),suppose(A,F)).
precond(suppose(A,F),C) :- property(F),abd(A,F,C).
template(suppose(A,F),[A,' supposes that '|Ft]) :-
  def_template(F,A,nil,Ft).

abd(A,Fr,Fc) :-
  inf_rule(A,P=>F),
  on_conj(Fr,P),
  (believes(A,F),!,Fc = true;
   Fc = /believes(A,F);
   Fc = sensed(A,F);
   Fc = told(_,[A,F])).

% operation that changes the current place of a character

operation(go(C,L1,L2)).
deleted(current_place(C,L1),go(C,L1,L2)).
added(current_place(C,L2),go(C,L1,L2)).
precond(go(C,L1,L2),
  (city(L1),city(L2))).
template(go(C,L1,L2),[C,' goes from ',L1,' to ',L2]).


/* PART 3 - facilities to handle facts */

% listing all property-denoting facts holding at the current state

facts :-
  nl,
  forall((property(X),X),describe(X)), nl.

property(not F,P,C) :-
  property(F,P,C).

property(F,P,C) :-
  fact(F),
  F =.. [P,C],
  entity(P,_).
property(F,P,C) :-
  fact(F),
  F =.. [P,C,_],
  attribute(_,P),
  not member(P,
   [believes,will_tell,told,will_ask,asked,sensed,watched,inferred,supposed]).
property(F,P,C) :-
  fact(F),
  (F =.. [P,C,_]; F =.. [P,_,C]),
  relationship(P,_),
  not P = trusts.

property(F) :-
  fact(F),
```

```
  one(property(F,_,_)).

% listing the information-package facts holding at the current state

info_facts :-
  nl,
  forall(
   (fact(X),
   not property(X,_,_),
   X,
   clause(X,true)),
   (template(X,T),
    xclist(T,L),
   write('    '),write(L),nl)),
   nl.

template(told(A,[B,F]),T) :-
  template(tell(A,B,F),T1),
  replace(' tells ',' told ',T1,T).
template(asked(A,[B,F]),T) :-
  template(ask(A,B,F),T1),
  replace(' asks ',' asked ',T1,T).
template(sensed(A,F),T) :-
  template(sense(A,F),T1),
  replace(' senses that ',' sensed that ',T1,T).
template(watched(A,E),T) :-
  template(watch(A,E),T1),
  replace(' watches the event: \'',' watched the event: \'',T1,T).
template(inferred(A,F),T) :-
  template(infer(A,F),T1),
  replace(' infers that ',' inferred that ',T1,T).
template(supposed(A,F),T) :-
  template(suppose(A,F),T1),
  replace(' supposes that ',' supposed that ',T1,T).
template(believes(A,F),[A,' believes that '|Ft]) :-
  def_template(F,A,nil,Ft).
template(trusts(A,B),[A,' trusts ',B]).


/* PART 4 - template definition */

def_template(F,C1,C2,T) :-
  (F = (not Ft), !, M = neg;
   Ft = F, M = pos),
  def_template(Ft,M,C1,C2,T).

def_template(F,M,C1,C2,T) :-
  F =.. [P,Cf,V],!,
  (C2 = nil,!,
    (C1 = Cf,!,
     gender(C1,G),
     (G = male,!, C3 = he;
     G = female, C3 = she);
    C3 = Cf);
   C3 = Cf),
  Ft =.. [P,C3,V],
  f_template(Ft,Ts),
  (M = pos, !,
    (C1 = C3, !, Mt = i_pos;
    C2 = C3, !, Mt = y_pos;
    Mt = pos);
   M = neg, !,
    (C1 = C3, !, Mt = i_neg;
```

```
    C2 = C3, !, Mt = y_neg;
    Mt = neg);
   M = int, ground(F), !,
    (C1 = C3, !, Mt = i_int;
    C2 = C3, !, Mt = y_int;
    Mt = int);
   M = int, not ground(F), !,
    (var(C3), !, Mt = int_who;
     C1 = C3, !, Mt = i_int_v;
    C2 = C3, !, Mt = y_int_v;
    Mt = int_v)),
  member(Mt:T,Ts).

def_template(F,M,C1,C2,T) :-
  F =.. [P,Cf],
  (C2 = nil,!,
    (C1 = Cf,!,
     gender(C1,G),
     (G = male,!, C3 = he;
     G = female, C3 = she);
    C3 = Cf);
   C3 = Cf),
  Ft =.. [P,C3],
  f_template(Ft,Ts),
  (M = pos, !,
    (C1 = C3, !, Mt = i_pos;
    C2 = C3, !, Mt = y_pos;
    Mt = pos);
   M = neg, !,
    (C1 = C3, !, Mt = i_neg;
    C2 = C3, !, Mt = y_neg;
    Mt = neg);
   M = int, ground(F), !,
    (C1 = C3, !, Mt = i_int;
    C2 = C3, !, Mt = y_int;
    Mt = int);
   M = int, not ground(F), !,
     Mt = int_v),
  member(Mt:T,Ts).

f_template(watched(X,E),
  [pos: [X, ' watched the event: '''|Et],
   i_pos: ['I watched the event: '''|Et],
   y_pos: ['You watched the event: '''|Et],
   neg: [X, ' did not watch the event: '''|Et],
   i_neg: ['I did not watch the event: '''|Et],
   y_neg: ['You did not watch the event: '''|Et],
   int: ['Did ',X,' watch the event: '''|Et],
   i_int: ['Did I watch the event: '''|Et],
   y_int: ['Did you watch the event: '''|Et],
   int_v: ['Who did watch the event: '''|Et]]) :-
     template(E,Et1),
    varnames(Et1),
    append(Et1,[''''],Et).


/* PART 5 - higher-level facilities */

survey(A,F,F:S) :-
  listvar(F,Lv),
  setof(P:F,
    (T,B,Plan,Lv)^(plans(believes(A,F),Plan), s_added(Plan,T),member(P,T),
      (P = sensed(A,F); P = inferred(A,F); P = supposed(A,F); P = told(B,[A,F])))),
```

```
   S).

survey_v(A,F,F:S) :-
  listvar(F,Lv),
  setof(P:F,
    (T,B,Plan,Lv)^(plans((believes(A,F),valid(F)),Plan),
s_added(Plan,T),member(P,T),
      (P = sensed(A,F); P = inferred(A,F); P = supposed(A,F); P = told(B,[A,F]))),
   S).

rank(A,F:S,Sr) :-
  weights(A,F,W),
  rank1(S,W,Sw),
  sort(Sw,S1),
  rank2(S1,S2),
  sort(S2,S3),
  reverse(S3,Sr).

rank1([],_,[]).
rank1([P:F|R],W,[F:Wi|S]) :-
  copy(W,Wc),
  member(Wi:P,Wc),!,
  rank1(R,W,S).
rank1([P:F|R],W,[F:0|S]) :-
  rank1(R,W,S).

rank2([],[]).
rank2([F:V1|R],[Vs:F|S]) :-
  rank2(R,[V2:F|S]),!,
  Vs is V1 + V2.
rank2([F:V|R],[V:F|S]) :-
  rank2(R,S).

violations(Fs,Vs) :-
  findall(Rn,(violate_rule(Rn,Fs,C),C),Vs).

valid(Fs) :-
  violations(Fs,[]).

recognize_lib(Obs,Pl,Lib) :-
  member(Pl,Lib),
  recognize(Obs,Pl).

recognize(Obs,Pl) :-
  ex1p(Pl,L),
  recog(Obs,L,Rec),
  chk_patt(Rec,Obs).

recog([],_,[]).
recog([X1|R],[X2|L],[X2|T]) :-
  copy(X2,X3),
  X1 = X3,
  recog(R,L,T).
recog([X|R],[_|S],T) :-
  recog([X|R],S,T).

check_obs(Obs,Pl) :-
  ex1p(Pl,L),
  check_obs1(Obs,L,Rec),
  chk_patt(Obs,Rec).

check_obs1([],_,[]).
check_obs1([X1|R],[X2|L],[X2|T]) :-
```

```
  copy(X1,X3),
  X2 = X3,
  check_obs1(R,L,T).
check_obs1([X|R],[_|S],T) :-
  check_obs1([X|R],S,T).


/* PART 6 - utilities */

replace(_,_,[],[]) :- !.
replace(X,Y,X,Y) :-
  atomic(X), !.
replace(X,Y,X,Y) :-
  var(X), !.
replace(X,Y,Z,Z) :-
  atomic(Z), !,
  not (X = Z), !.
replace(X,Y,[Z|L],[Z1|L1]) :- !,
  replace(X,Y,Z,Z1),
  replace(X,Y,L,L1).
replace(X,Y,Z1,Z2) :-
  Z1 =.. [F|L],
  replace(X,Y,F,F1),
  replace(X,Y,L,L1),
  Z2 =.. [F1|L1].

chk_patt(P,L) :-
  listvar(P,Lv),
  count(Lv,N),
  P = L,
  xsetof(X,member(X,Lv),Lvc),
  count(Lvc,N).

iterate(N,X) :-
  it_i,
  X,
  it(N), !,
  retract(it_m(_)).

it_i :-
  (it_m(X), retract(it_m(X));
   true),
  assert(it_m(1)).

it(N) :-
  it_m(I),
  (I = N,!;
   J is I+1,
   retract(it_m(I)),
   assert(it_m(J)), !,
   fail).

clear :-
  findall(H/N,(fact(F),F=..[H|_],current_predicate(H/N)),S),
  forall(member(H/N,S),abolish(H/N)),
  retractall(log(_)),
  retractall(pre_state(_,_)).

reconsult(P) :-
  clear, consult(P).
```

```
/* ======================================================================== */
/*                           EXAMPLE                                        */
/* ======================================================================== */

/* PART 7 - example static schema */
/* also contains additional properties for the 'person' entity */

:- dynamic country/1,city/1,hair_colour/2,daltonic/2,born/2,home/2,citizen/2.

entity(country,country_name).
entity(city,city_name).
attribute(person,hair_colour).
attribute(person,daltonic).
relationship(born,[person,country]).
relationship(home,[person,country]).
relationship(citizen,[person,country]).

taken_as_place(city).


/* PART 8 - templates for the properties */

f_template(person(X),
  [pos: [X,' is a person'],
   i_pos: ['I am a person'],
   y_pos: ['You are a person'],
   neg: [X,' is not a person'],
   i_neg: [X,'I am not a person'],
   y_neg: ['You are not a person'],
   int: ['Is ',X,' a person'],
   i_int: ['Am I a person'],
   y_int: ['Are you a person'],
   int_v: ['Who is a person']]).

f_template(country(X),
  [pos: [X,' is a country'],
   neg: [X,' is not a country'],
   int: ['Is ',X,' a country'],
   int_v: ['What country is there']]).

f_template(city(X),
  [pos: [X,' is a city'],
   neg: [X,' is not a city'],
   int: ['Is ',X,' a city'],
   int_v: ['What city is there']]).

f_template(gender(X,Y) ,
  [pos: [X,' is ',Y],
   i_pos: ['I am ',Y],
   y_pos: ['You are ',Y],
   neg: ['You are not ',Y],
   i_neg: ['I am not ',Y],
   y_neg: ['You are not ',Y],
   int: ['Is ',X,' ',Y],
   i_int: ['Am I ',Y],
   y_int: ['Are you ',Y],
   int_v: ['Is ',X,' male or female'],
   i_int_v: ['What am I, male or female'],
   y_int_v: ['What are you, male or female'],
   int_who: ['Who is ',Y]]).

f_template(hair_colour(X,Y),
  [pos: [X,' has ',Y,' hair'],
```

```
   i_pos: ['My hair is ',Y],
   y_pos: ['Your hair is ',Y],
   neg: [X,' has not ',Y,' hair'],
   i_neg: ['My hair is not ',Y],
   y_neg: ['Your hair is not ',Y],
   int: ['Has ',X,' ',Y,' hair'],
   i_int: ['Have I ',Y,' hair'],
   y_int: ['Is your hair ',Y],
   int_v: ['What is the colour of the hair of ',X],
   i_int_v: ['What is the colour of my hair'],
   y_int_v: ['What is the colour of your hair'],
   int_who: ['Who has ',Y,' hair']]).

f_template(daltonic(X,Y),
  [pos: [X,' is daltonic'],
   i_pos: ['I am daltonic'],
   y_pos: ['You are daltonic'],
   neg: [X,' is not daltonic'],
   i_neg: ['I am not daltonic',Y],
   y_neg: ['You are not daltonic',Y],
   int: ['Is ',X,' daltonic'],
   i_int: ['Am I daltonic'],
   y_int: ['Are you daltonic'],
   int_v: ['Is ',X,'daltonic or not'],
   i_int_v: ['Am I daltonic or not'],
   y_int_v: ['Are you daltonic or not'],
   int_who: ['Who is daltonic']]).

f_template(born(X,Y),
  [pos: [X,' was born in ',Y],
   i_pos: ['I was born in ',Y],
   y_pos: ['You were born in ',Y],
   neg: [X,' was not born in ',Y],
   i_neg: ['I was not born in ',Y],
   y_neg: ['You were not born in ',Y],
   int: ['Was ',X,' born in ',Y],
   i_int: ['Was I born in ',Y],
   y_int: ['Were you born in ',Y],
   int_v: ['Where was ',X,' born'],
   i_int_v: ['Where was I born'],
   y_int_v: ['Where were you born'],
   int_who: ['Who was born in ',Y]]).

f_template(home(X,Y),
  [pos: [X,' lives in ',Y],
   i_pos: ['I live in ',Y],
   y_pos: ['You live in ',Y],
   neg: [X,' does not live in ',Y],
   i_neg: ['I do not live in ',Y],
   y_neg: ['You do not live in ',Y],
   int: ['Does ',X,' live in ',Y],
   i_int: ['Do I live in ',Y],
   y_int: ['Do you live in ',Y],
   int_v: ['Where does ',X,' live'],
   i_int_v: ['Where do I live'],
   y_int_v: ['Where do you live'],
   int_who: ['Who lives in ',Y]]).

f_template(citizen(X,Y),
  [pos: [X,' is a citizen of ',Y],
   i_pos: ['I am a citizen of ',Y],
   y_pos: ['You are a citizen of ',Y],
   neg: [X,' is not a citizen of ',Y],
```

```
   i_neg: ['I am not a citizen of ',Y],
   y_neg: ['You are not a citizen of ',Y],
   int: ['Is ',X,' a citizen of ',Y],
   i_int: ['Am I a citizen of ',Y],
   y_int: ['Am I a citizen of ',Y],
   int_v: ['Of what country is ',X,' a citizen'],
   i_int_v: ['Of what country am I a citizen'],
   y_int_v: ['Of what country are you a citizen'],
   int_who: ['Who is a citizen of ',Y]]).

f_template(current_place(X,Y),
  [pos: [X,' is now in ',Y],
   i_pos: ['I am now in ',Y],
   y_pos: ['You are now in ',Y],
   neg: [X,' is not in ',Y,' now'],
   i_neg: ['I am not now in ',Y,' now'],
   y_neg: ['You are not now in ',Y,' now'],
   int: ['Is ',X,' in ',Y,' now'],
   i_int: ['Am I in ',Y],
   y_int: ['Are you in ',Y],
   int_v: ['Where is ',X,' now' ],
   i_int_v: ['Where am I'],
   y_int_v: ['Where are you'],
   int_who: ['Who is now in ',Y]]).


/* PART 9 - conditioners */

% rules: will_tell

will_tell('John','Peter',F,believes('John',F)).
will_tell('John','Mary',F,(copy(F,F1),believes('John',F),asked('Mary',['John',F1]
))).
will_tell('John','Mary',born(X,L),(believes('John',born(X,L1)),country(L),not (L
== L1),not (X == 'Mary'))).
will_tell('Peter','Mary',F,(copy(F,F1),believes('Peter',F),asked('Mary',['Peter',
F1]))).
will_tell('Peter','Mary',watched('Peter',O),watched('Peter',O)).
will_tell('Mary','Laura',citizen('Mary',C),believes('Mary',citizen('Mary',C))).
will_tell('Laura','Peter',F,(property(F,P,'Laura'),F)).
will_tell('Laura','Peter',hair_colour('Mary',blond),true).
will_tell('Laura','Peter',citizen('Mary','London'),true).

% rules: will_ask

will_ask('Mary','John',current_place(P,L),person(P)).
will_ask('Mary','Peter',hair_colour('Mary',C),true).

% rules: regular sense_rules

sense_rule(W,F,true) :-
  not (daltonic(W,true), (F = hair_colour(_,_); F = (not hair_colour(_,_)))).

% rules: distorted sense_rules

sense_rule(W,hair_colour(P,C1),hair_colour(P,C2),(hair_colour(P,C1),map_dalt(C1,C
2))) :-
  daltonic(W,true).
sense_rule(W,not hair_colour(P,C1),not hair_colour(P,C2),
           (hair_colour(P,C1x),map_dalt(C1x,C2x),not (C2x =
C2),map_dalt1(C2,C1,C1x))) :-
  daltonic(W,true).
```

```
map_dalt(C1,C2) :-
  (C1 = red, !, C2 = green;
   C1 = green, !, C2 = red;
   C2 = C1).

map_dalt1(C1,C2,C3) :-
  map_dalt(C1,Cx),
  (C1 = Cx, !;
   not (C1 = C3), !;
   C2 = Cx).

% rules: watch_rules

watch_rule(C,go(A,L1,L2),true,L1).
watch_rule(C,go(A,L1,L2),true,L2).

% rules: inf_rules

inf_rule(A,(born(X,C),home(X,C)) => citizen(X,C)) :- person(A).
inf_rule(A,(daltonic(B,true),told(B,[A,hair_colour(P,C1)]),map_dalt(C1,C2), not
(C1 == C2)) =>
  hair_colour(P,C2)).
inf_rule(C,(watched(C,go(A,L1,L2)) => (not (current_place(A,L1))))).
inf_rule(C,(watched(C,go(A,L1,L2)) => (current_place(A,L2)))).


/* PART 10 - clauses used by the higher-level facilities */

% ranking weights

weights('Peter',hair_colour(X,Y),[1:sensed('Peter',_),2:told('John',['Peter',_])]
).
weights('Peter',born(X,Y),[2:sensed('Peter',_),1:supposed('Peter',_),4:told('John
',['Peter',_])]).
weights('Peter',citizen(X,Y),[2:sensed('Peter',_),1:supposed('Peter',_),4:told('J
ohn',['Peter',_])]).

% violations

violate_rule(r1,Cs,(on_conj(citizen(X,Y),Cs), (not person(X); not country(Y)))).
violate_rule(r2,Cs,(setof(C,on_conj(citizen(X,C),Cs),S),count(S,N),N > 1)).

% plan library */

lib([
  start=>go(A,L1,L2)=>go(B,L1,L2)=>sense(A,current_place(B,L2)),
  start=>go(A,L1,L2)=>go(A,L2,L1)
  ]).


/* PART 11 - initial state */

person('John').
person('Mary').
person('Peter').
person('Laura').
country('UK').
country('USA').
country('Canada').
city('London').
city('Manchester').
gender('Mary','female').
gender('Laura','female').
```

```
gender('John','male').
gender('Peter','male').
home('Mary','UK').
born('Mary','UK').
born('Peter','UK').
born('Laura','USA').
citizen('Laura','Canada').
current_place('John','London').
current_place('Peter','London').
current_place('Mary','London').
current_place('Laura','Manchester').
daltonic('Peter',true).
hair_colour('Mary',red).
hair_colour('Laura',blond).

trusts('Peter','John').
trusts('Mary','John').
trusts('Peter','Laura').
trusts('Laura','Mary').

believes('John',F) :- property(F), F.
believes(X,F) :- person(X), not (X == 'John'), fact(F), property(F,_,X),F.
```