PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 05/15

A STORY-BASED APPROACH TO INFORMATION SYSTEMS

Vinicius M. Gottin
Marco A. Casanova
Edirlei Soares de Lima
Antonio L. Furtado

Departamento de Informática

Monografias em Ciência da Computação, No. 05/15 ISSN 0103-9741
Editor: Prof. Carlos José Pereira de Lucena November, 2015

A Story-Based Approach to Information Systems

Vinicius M. Gottin[1], Marco A. Casanova[1], Edirlei Soares de Lima[2], Antonio L. Furtado[1]

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

Abstract: We claim that characterizing information system domains by what stories can emerge from their formal specification helps determining whether a proposed rule-based conceptual modelling operational definition of the domain meets the prospective users' expectations. To provide a suitable environment, we start with a running specification, where event-producing operations are conceptually defined in declarative style in terms of their pre-conditions and post-conditions, and plan generation is employed to compose story-plots, whose execution is simulated in workspace memory. This three-schemata conceptual specification, expressed in a logic programming formalism, is processed along successive stages towards a DBMS implementation. While plan generation remains available to examine alternative ways to reach future states, an event-oriented database log, which registers the executed event-producing operations with their timestamp and transaction reference number, provides a repository of past stories. To conduct experiments while investigating how to thus extend *data*-bases in the direction of intelligent *story*-bases, we developed a prototype tool, called IDB, which is now fully operational.

Keywords: Storytelling, Conceptual Specification, Entity-Relationship Model, Relational Databases, Plan Generation, Simulation, Logic Programming, Plot-Mining.

Resumo: Afirmamos que a caracterização de domínios de sistemas de informação pelas estórias que podem emergir de sua especificação formal ajuda a determinar se uma dada definição operacional, proposta através de modelagem conceitual baseada em regras, satisfaz as expectativas dos usuários em perspectiva. Com o objetivo de prover um ambiente adequado, partimos de uma especificação executável, onde operações produtivas de eventos são definidas conceitualmente em estilo declarativo em termos de suas pré-condições e pós-condições, e se emprega geração de planos para compor enredos de estórias, cuja execução é simulada na memória de trabalho. Essa especificação conceitual em três esquemas, expressa em um formalismo de programação em lógica, é processada ao logo de sucessivos estágios até ser implementada por meio de algum SGBD. Enquanto a geração de planos permanece disponível para examinar meios alternativos de atingir estados futuros, um log de bancos de dados orientado para eventos – que registra a execução das operações produtivas de eventos, junto com seu selo temporal e número de referência da transação – serve de repositório para as estórias do passado. Para conduzir experimentos durante a investigação de como assim estender bancos-de-*dados* na direção de bancos-de-*estórias*, desenvolvemos um protótipo, de nome **IDB**, atualmente em pleno funcionamento.

Palavras-chave: Narração de estórias, Especificação Conceitual, Modelo Entidades-Relacionamentos, Bancos de Dados Relacionais, Geração de Planos. Simulação, Programação em Lógica, Mineração de Enredos.

1. Introduction

When elaborating a formal specification, designers always strive to make sure that all data integrity constraints and business norms, regulating the legitimate processes and barring unauthorized conduct, are enforced. Yet, once the specification is formulated as a set of interrelated logic rules, it is hard to predict what situations can result.

This work proposes to view information system domains in terms of the *stories* emerging from their formal specification. As Schank asserts [1995], human intelligence is led in a very high degree by the stories in which the person has participated in some role, or has heard from other persons.

Our approach to, so to speak, upgrade *data*-bases to *story*-bases, relies on an environment offering, at least:

- the *vocabulary* of the stories, consisting of a set of conceptually specified event-producing operations;
- a *textual repository* of the story events, under the form of a time-stamped `LOG` of the event-producing operations executed;
- a *plot-composition* device, to plan sequences of event-producing operations, purporting to satisfy the goals of the authorized agents.

A domain-oriented vocabulary lends semantic significance to the state transitions occurring in the system's lifetime. Instead of restricting the attention to detailed insertions, deletions or updates of physical records, one can, in an academic domain for example, consider events such as `offer` (a course), `enroll` (a student in a course), etc. In turn, an event-structured `LOG` meaningfully portrays the stories that happened in the *past*, whereas plan-generation allows to examine alternative *future* stories.

Building the proposed environment for a given information system domain involves going all the way from conceptual specification, in a Logic Programming formalism appropriate to conduct simulation runs, to implementation in a relational Database Management System (DBMS). In this paper, we describe such an environment both abstractly and as made to work in a prototype, named **IDB**, built to illustrate our proposal.

Section 2 positions our proposal regarding related work. Section 3 reviews our conceptual specification discipline, wherein not only the facts that shall constitute the database, but also events and agents are modelled, and plan-generation makes the specification executable. Section 4 treats the stepwise transition from workspace to DBMS environment. Section 5 deals with **IDB,** showing how it performs such tasks as plot generation, finding recurring patterns, and recovering past stories from an event-oriented `LOG`. Section 6 contains concluding remarks. The full conceptual specification and the main components of its implementation in the DBMS environment are reproduced in the Appendix.

2. Related Work

The formalization of stories as a research field dates back to seminal works based on literary theory [Propp 1968] and the philosophy of language and cognition [Charniak 1972, Prince 1973]. Early story generation programs aiming at the understanding of narrative structure like TALE-SPIN [Meehan 1981] and, in the following decade, [Brewer 1980, Lang 1999] enabled and gave rise to works that applied automated planning to compose narratives [Mateas 2003, Riedl 2004].

The conceptual specification in a logic programming formalism featured in this paper follows [Furtado 2000], where plots, under the form of plan-generated sequences of events, provide an operational characterization of narrative genres. Our work is closely related to previous research on narrative generation based on automated planning techniques (as surveyed by Barros [2007]), as well as on the representation of actions and hypothetical situations [Miller 1994]. Also, the provision of *running* conceptual model specifications associates our proposal with *event simulation* [Ciarlini 2002], inasmuch as it allows checking whether the behaviour of the system corresponds to the designer's expectations.

Some of the tasks involving our `LOG` of semantically meaningful operations (cf. section 5) – displaying past stories and retrieving past database events – relate to the topic of *temporal databases* [Date 2002].

Other tasks, such as finding patterns by comparing story sequences or the statistical analysis of the `LOG`, relate to *process mining* for knowledge discovery [Aalst 2011], aiming, in particular, at the enhancement of the system's design.

3. Running conceptual modelling specification

The starting point of our long-term project was the development of running conceptual level specifications to characterize narrative genres in a logic programming formalism [Furtado 2000].

To specify an information system application, it is not enough to define the classes of *facts* that will eventually populate the underlying database. One should specify, also in conceptual terms, i.e. in the language of the application domain, a fixed repertoire of *events*, whereby the state of the mini-world would change. And a pragmatic aspect, which has to do with how the *agents* involved would be motivated to reach their goals by bringing about the appropriate events, should also be considered (for a comprehensive formal discussion, cf. [Ciarlini 2010]). Facts, events and agents are contemplated, respectively, in what we call *static*, *dynamic* and *behavioural* schemas.

3.1 The Static Schema

The *static schema*, wherein *facts* are specified in the Entity-Relationship model [Batini 1991], defines the entity classes (e.g. `student)` and their identifying attributes (e.g. `student_name`). Entities can have additional attributes (e.g. `credits`, for the `course` entity). Relationships associate entities (e.g. `takes` associates `student` and `course` entities). One-to-n binary relationships are declared in `f_relationship` clauses, where "f" stand for "functional".

The complete list of clauses in the static schema of the example used throughout this paper follows below:

```
entity(student,student_name).
attribute(student,credits_won).
entity(program,program_name).
attribute(program,requirement).
entity(course,course_name).
attribute(course,credits).
entity(teacher,teacher_name).
attribute(teacher,position).
relationship(takes,[student,course]).
attribute(takes,grade).
relationship(has_finished,[student,course]).
relationship(has_dropped,[student,course]).
relationship(graduated_in,[student,program]).
f_relationship(taught_by,[course,teacher]).
attribute(taught_by,textbook).
```

An instantiation of the schema, representing a series of facts in clause format, can be supplied to express an *initial state*. Thus `program('Alpha')` indicates the existence of a program entity with `program_name` "Alpha" as identifying attribute. A separate attribute clause, `requirement('Alpha',5)`, would further characterize "Alpha" as requiring a total of 5 credits.

Any application domain is subject to *integrity constraints*, some of which are specific to the domain, whereas others are inherent in the Entity-Relationship model. One of the latter is that instances of attributes and relationships can only exist while the entity instances of which they are properties exist. This constraint, like all others, must prevail in the initial state and must be enforced whenever state changes affecting the entity instances occur. Typical policies to prevent integrity violations are: (**a**) *reject* updates resulting in "dangling" attribute or relationship instances, e.g. do not delete a course in which there are students enrolled, or (**b**) *propagate* such updates, e.g. if a course is deleted also delete all student enrollments in the course. Our method to handle integrity constraints is to embed them in the definition of the chosen event-producing operations and only permit updates through these operations. The *dynamic schem*a, to be described next, concerns this orientation, whereby a sound *abstract data type* [Guttag 1977] discipline is imposed.

3.2 The Dynamic Schema

The *dynamic schema* deals with *events* able to change the state of the mini-world of the application domain. They are limited to a repertoire of operations, defined by their pre-conditions and post-conditions (effects), following the STRIPS formalism [Fikes 1971], a representation that lends itself to the application of plan generation algorithms, as described in section 3.4.

The event-producing operations provided for the academic example used to illustrate the discussion are informally described below:

- operation offer(C,N,T,B) - offer course C, with N credits, taught by teacher T, using textbook B
pre-conditions: course C is not offered, even with a different number of credits, and T is already a teacher
post-conditions: clauses course(C), credits(C,N), taught_by(C,T), and textbook(C,T,B) are added

- operation hire(T,P) - hire teacher T as a (P-)professor, where position P designates an academic level, such as assistant, associate, full, etc.
pre-conditions: none
post-conditions: clauses teacher(T) and position(T,P) are added

- operation enroll(S,C) - enroll student S in course C
pre-conditions: course C is currently offered, and student S has neither finished nor dropped it before
post-conditions: clauses takes(S,C) and grade(S,C,'inc') – where 'inc' stands for incomplete – are added; the clauses student(S) and credits_won(S,0) are added only if this is the first enrollment of S in a course

- operation transfer(S,C1,C2) - transfer student S from course C1 to course C2
pre-conditions: course C2 is currently offered, student S is taking C1 and has neither finished nor dropped C2 before
post-conditions: clauses takes(S,C1) and grade(S,C1,G) are deleted and clauses dropped(S,C1), takes(S,C2) and grade(S,C2,'inc') are added
- operation drop(S,C) - drop the enrollment of student S in course C
pre-conditions: none
post-conditions: clauses takes(S,C) and grade(S,C,G) are deleted and clause has_dropped(S,C) is added

- operation cancel(C) - cancel course C
pre-conditions: no student is currently taking C
post-conditions: clauses course(C) and credits (C,N) are deleted

- operation change_cr(C,N1,N2) - change the number of credits of course C from N1 to N2
pre-conditions: C is being offered with N1 credits
post-conditions: clause credits(C,N1) is deleted and clause credits(C,N2) is added

- operation mark(S,C,G) - mark with grade G the performance of student S in course C
pre-conditions: the current grade of S in C is 'inc' (incomplete)
post-conditions: clause grade(S,C,'inc') is deleted and clause grade (S,C,G) is added

- operation pass(S,C,T1,T2) - let student S pass course C, thereby increasing the total number of credits from T1 to T2
pre-conditions: S is taking C, which gives N credits, has obtained a 'pass' grade in the course, and until then has completed T1 credits; T2 is obtained by summing up N to T1
post-conditions: clauses takes(S,C), grade (S,C,G) and credits_won(S,T1) are deleted, and clauses credits_won(S,T2) and has_finished(S,C) are added

- operation create_program(P,R) - create program P with the requirement of R credits
pre-conditions: program P is not already being offered, with any number of total credits required
post-conditions: clauses program(P) and requirement(P,R) are added

- operation receive_degree(S,P) - let student S graduate in program P
pre-conditions: student S has obtained exactly the total number of credits required for the completion of program P and is no longer taking courses
post-conditions: clause graduated_in(S,P) is added

Every operation is defined in clausal format. The definition of `enroll(S,C)` is shown below, and for the other operations we refer the reader to the source document provided for the full specification.

```
operation(enroll(S,C)).
/added(student(S),enroll(S,C)).
/added(credits_won(S,0),enroll(S,C)) :-
  not credits_won(S,_).
added(takes(S,C),enroll(S,C)).
added(grade(S,C,'inc'),enroll(S,C)).
precond(enroll(S,C),
  (course(C),
   /(not has_finished(S,C)),
   /(not has_dropped(S,C)))).
```

3.3 The Behavioural Schema

The **behavioural schema** concerns the authorized *agents*, who are motivated to perform events in order to achieve goals induced by certain situations, as expressed in situation-objective (`sit_obj`) rules. When the planner is applied to these rules, alternative future stories are composed, as shown in section 3.4. In [Barbosa 2015] we considered an extension to the schema, that purports to model the agents' personality in terms of drives, attitudes and emotional profile.

In the rule below the intended agent is a student `S`. The situation is: student `S` has dropped course `C1`, which gives `Cr1` credits, and is currently taking no course; there exists, on the other hand, a course `C2` with a smaller number of credits, `Cr2`, which is therefore presumed to be easier than `C1`. The goal is that `S` would be advised to take `C2`.

```
sit_obj(student(S),
  ( has_dropped(S,C1),
    credits(C1,Cr1),
    not takes(S,Cx),
    course(C2),
    credits(C2,Cr2),
    Cr2 < Cr1),
  takes(S,C2)).
```

A second rule, where the Chairman is the agent, refers to the critical situation of a course that was created two or more years ago and that no student has passed until now. If this occurs for some course `C`, the recommended goal is that `C` should cease to be offered.

```
sit_obj(chairman,
  ( course(C),
    select_log(R,Ts,offer(C,Cr)),
    in_timestamp(year,Yevent,Ts),
    in_current_time(year,Ynow),
    Diff is Ynow - Yevent,
    Diff >= 2,
    not select_log(_,_,pass(_,C,_,_))),
   not course(C)).
```

A distinctive characteristic of this rule is that it requires inspection of the database `LOG` table, and consequently assumes that the system has been implemented – a topic to be treated in section 4.

3.4 Plan Generation Features

Our planning algorithms enforce a discipline that in most cases simplifies the definition of the operations. First of all, it is not necessary (although it may be done, in order to guarantee the instantiation of certain parameters) to declare as a pre-condition that a fact to be added by an operation must not already hold at the current state, or that a fact to be deleted does not hold.

Moreover, an operation is caused to fail if for any reason it cannot produce all the specified effects, except for those that, in the clausal notation, are prefixed with a `/`. This notation indicates that an effect of the form `/F` (or `/(not F)`) will be performed "if needed"; so if `F` already holds (or does not hold) no failure will result. Consider again operation `enroll`: the effects (post-conditions) are that clauses `takes(S,C)` and `grade(S,C,'inc')` are always added, but `student(S)` and `credits_won(S,0)`, marked with `/`, are added only if this is the first enrollment of `S`.

The planner interprets the pre-conditions both as tests for the applicability of an operation `Op` and, in case of failure, as sub-goals to be fulfilled by operations to be inserted before `Op` in the plan. In the enroll operation, if course `C` is not offered, *its creation becomes a sub-goal*, which means that course offerings are sensitive to demand. This recursive treatment of pre-conditions as sub-goals constitutes the *backward chaining* strategy, on which many planning algorithms (including ours) are based.

However not all failed pre-conditions are converted into sub-goals. A second use of the `/` notation is to mark pre-conditions that are to be handled exclusively as tests: if a positive `/F` or negative `/(not F)` pre-condition for an operation fails, the operation is rejected. For example, operation `enroll` requires the student to not have dropped the course before; and if that is not the case *no sub-goal is created* and the operation fails. Hence, marked and unmarked pre-conditions reflect, respectively, policies **(a)** (*reject*) and **(b)** (*propagate*) exposed at the end of section 3.1.

An especially powerful feature that enables the planner to pursue goals involving numerical expressions is *constraint programming* [Rossi 2006]. Constraint programming also conveniently relies on delayed evaluation, waiting until all variables have been instantiated before proceeding to compute a formula such as T2 #= T1 + N, T1 #>= 0 – a feature employed in our `pass` operation in which the number of credits of a course is added to the previous total of credits obtained by the student. With Prolog alone it would not be possible to determine, with a single call to the planner, all events that student Bea should go through to successfully fulfill the requirement of program Alpha:

```
:- plans(graduated_in('Bea','Alpha'),P).
P = start=>enroll(Bea,Art)=>enroll(Bea,Semiotics)=>
mark(Bea,Semiotics,pass)=>pass(Bea,Semiotics,0,3)=>mark(Bea,Art,pass)=>pass(Bea,Art,3,5)=>receive_deg
ree(Bea,Alpha) .
```

Another special feature, needed to specify repetitive pre-conditions, directs the planner to consider, when given an expression of the form `T : I`, all terms `T` satisfying an *iterator* expression `I`. For example, the pre-condition of the operation `cancel(C)` that cancels a course is that no student should be taking it, which requires that the planner must – as an iterative sub-goal – retrieve and find *how to undo each current enrollment*, which is thus expressed:
`not takes(S,C):(student(S),takes(S,C))`

4. From workspace to database environment

Once the schemas have been specified and an *initial state* has been provided in workspace memory, consisting of ground clauses representing instances of the specified entity and relationship classes and their attributes, it is possible to achieve state transitions by executing – in the workspace – the clause additions and deletions defined as the effect of the event-producing operations. For example, the execution of `enroll('Bea','Art')` would have the effect of adding the clause `takes('Bea','Art')` and, since this would be her first enrollment, of adding clauses to register her as `student`, with zero `credits_won`. We refer to this as the *workspace* stage.

At a second, *mixed*, stage, one keeps issuing commands in logic programming notation, but their effects are redirected to operate over an actual database, handled by a relational DBMS. The first step towards this stage is to configure as relational tables the entities, attributes and relationships of the static schema.

In addition to the tables that store the factual data, two auxiliary tables are needed: the `REF` table and the `LOG` table. The `REF` table keeps the granted *references* that serve as identifiers for *transactions* composed of any number of events, which may be intercalated with events of other transactions and resumed in future sessions. The `LOG` table registers the execution of each event in a record containing the reference of the transaction, the timestamp read from the system's clock, and the name and parameter list of the operation executed.

The mixed stage also requires a mechanism that enables the specified operations to act upon and update these relational tables. This mechanism should enforce the correct usage of the `REF` and `LOG` tables in keeping record of the operations.

The first two stages are intended for the specification tasks and for performing simulation runs, guided by the plan-generator. At the third stage, the *deploy* stage, operations should take place directly in the database environment, through in-database execution of procedures equivalent to the operations. In this stage, the system would be ready for routine operational usage, employing a commercially available DBMS, coupled with a suitable host language.

5. Experimenting with the IDB prototype

To test the feasibility of the proposed story-based approach, we developed the **IDB** (**I**ntelligent **D**ata**b**ases) prototype tool. Indeed, the logic-oriented language paradigm that we have been consistently following has been associated with intelligent databases, given that it "elevates the design and development of database applications to the level of declarative, knowledge-based specifications" [Zaniolo 1992]. Ironically, experimenting with the prototype was for us a source of surprise, since more than once the goals we submitted to the planner led to results contrary to what we anticipated, which proved in retrospect to be an "obvious" consequence of the specified rules. That this can occur even in the trivial scope of a toy example would seem to confirm the usefulness of automatic devices whose intelligence consists in the ability to play with a given set of rules in an inexorably systematic way.

In the following sections, we introduce **IDB** as an implementation of the three stages and the requirements associated, as described in section 4. The **IDB** prototype is implemented in SWI-Prolog, taking advantage of its ODBC (Open Database Connectivity) interface to communicate with an Oracle database.

5.1 IDB architecture

The **IDB** prototype straightforwardly implements the workspace stage in Prolog, since the static, dynamic and behavioural schemata definitions in logic programming formalism can be directly loaded into such a Prolog environment – in fact, the source file provided with the complete specification is a Prolog code file. Figure 1 represents this stage.

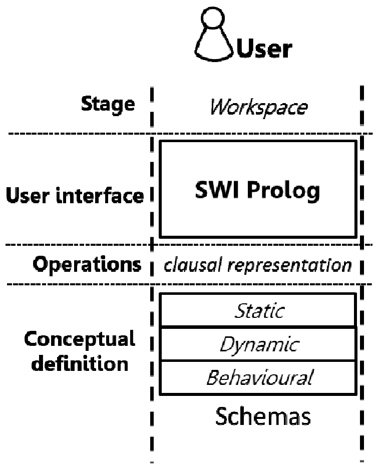


Figure 1: IDB's workspace stage architecture

For the mixed stage, **IDB** provides functionality for the setup of the required relational tables. The `gen_tabs` predicate creates separate tables for each entity class, with columns for their attributes, as well as the `REF` and `LOG` tables. As a customary optimization practice, additional columns are included to embed functional (i.e. one-to-n) binary relationships and their attributes. For example, given the constraint that a course is offered with just one appointed teacher, who adopts one textbook for the course, the following database command is generated to represent the `course` entity class jointly with the f-relationship `taught_by` in a single relational `COURSE` table:

```
CREATE TABLE "COURSE"
   ("COURSE_NAME" VARCHAR2(100),
    "CREDITS" NUMBER,
    "TEACHER_NAME" VARCHAR2(100),
    "TEXTBOOK" VARCHAR2(100)
   );
```

Separate tables are also created for non one-to-n binary relationships, with columns for their attributes. Relationships with more than two participating entities can be *reified*, i.e. treated as entities, with binary relationships leading to each participant.

IDB also provides the necessary mechanism for updating relational tables through the `compile_ops` predicate, which translates the original *declarative* specification of the event-producing operations into logic programming predicates exhibiting a *procedural* style. The pre-conditions are turned into calls to a Prolog-programmed `select` predicate that causes the DBMS to perform database `select` commands, and the post-conditions (additions and deletions) now take the form of `insert`, `delete` and `update` calls, equally programmed in Prolog.

We note, incidentally, that the argument passed in the calls to our `select` predicate can either refer directly to the relational tables, or, as needed in the compiled operations, to Entity-Relationship facts, which are internally translated into the appropriate `from` and `where` clauses. For instance, besides `select(course(C,N,T,B))`, which refers to an entire tuple, any of these calls are permitted:

```
select(course(C))
select(credits(C,N))
select(taugh_by(C,T))
select(textbook(C,T,B))
```

As an example of operation converted to procedural style, we show the compiled `enroll` predicate:

```
enroll(B, A) :-
        ref(C),
        select(course(A)),
        not select(has_finished(B, A)),
        not select(has_dropped(B, A)),
        (   select(student(B))
        ;   not select(student(B)),
            insert(student(B, 0))
        ),
        insert(takes(B, A, inc)),
        ins_log(C, enroll(B, A)).
```

After both the tables and the procedural operations have been created, a call to `env_option(db)` turns on a flag that directs the planner to no longer check each fact `F` by looking for clause `F` in the workspace, but rather by accessing the appropriate database table via a `select(F)` call. The architecture of **IDB** when acting on the mixed stage is shown in figure 2.

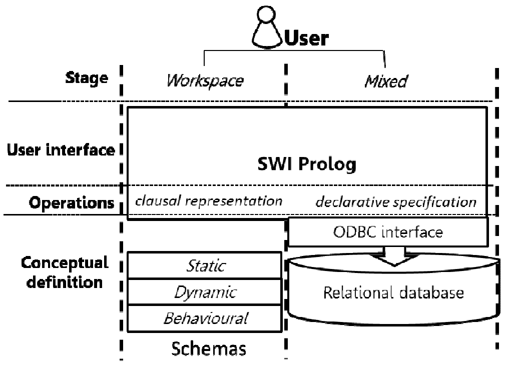


Figure 2: IDB's mixed stage architecture

Finally, **IDB** also implements a functionality to ease the transition to the third stage, deploy. With this objective, specifically, a second compiler (predicate `compile_proc`) translates from the procedural version of the operations, produced by the first compiler, into DBMS *stored procedures*:

```
CREATE PROCEDURE enroll(rn NUMBER, A VARCHAR2, B VARCHAR2) IS
 found NUMBER;
 ev VARCHAR(100);
 course_name_B VARCHAR(100);
BEGIN
 SELECT r INTO found FROM ref WHERE r = rn;
 SELECT course_name INTO course_name_B FROM
  course WHERE course_name = B;
 SELECT COUNT(*) INTO found FROM has_finished
  WHERE student_name = A AND course_name = B;
 IF found > 0 THEN RETURN; END IF;
 SELECT COUNT(*) INTO found FROM has_dropped
  WHERE student_name = A and course_name = B;
 IF found > 0 THEN RETURN; END IF;
 SELECT COUNT(*) INTO found FROM student
  WHERE student_name = A;
 IF found = 0 THEN INSERT INTO student
  VALUES(A, 0);
 END IF;
 INSERT INTO takes VALUES(A, B, 'inc');
 ev := 'enroll(' || A || ',' || B || ')';
 ins_log(rn,ev);
 COMMIT;
END enroll;
```

At the deploy stage, **IDB** allows to operate the database directly through SQL*Plus and the Oracle Database Xe shell. But even after reaching this final database stage, we find advisable to keep the logic programming specification alive, since it serves several practical purposes, including: documentation; training the users; further simulation and continuous testing; redesign; monitoring; data mining; and, especially, *plot mining* (cf. [Aalst 2011, Furtado 2007]). The deploy stage architecture is shown in figure 3.

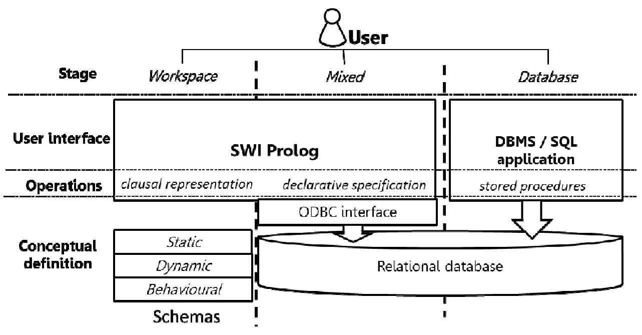


Figure 3: IDB's deploy stage architecture

5.2 Exploring the System on Hand with IDB

In terms of functionality, **IDB** enables to perform, among others, the following tasks, to be illustrated by example runs:

1. Extract and display past stories from the `LOG`.
2. Monitor the occurrence of situations that motivate the agents' goals.
3. Choose a plan to reach a goal.
4. Find patterns by comparing story sequences.
5. Apply statistical analysis to the `LOG`.
6. Retrieve past database states.
7. Schedule future events.

As an example of **Task 1** - *Extracting and displaying past stories from the* LOG, suppose we wish to concentrate on some element that may be involved in one or more events, such as the name "Bea", a model student with a highly praised performance, enough to obtain the total of 5 credits required by program Alpha. The query takes the form:

```
:- involved(log,'Bea',Story),show_events(Story).
```

and the answer is given in template-driven natural language, in an event-by-event narrative style that is adequate to stress temporal sequentiality:

```
Student Bea enrolled in course Art. Student Bea enrolled in course Semiotics. Bea's mark in
Semiotics was 'pass'. Student Bea, having passed course Semiotics, has a total of 3 credits.
Bea's mark in Art was 'pass'. Student Bea, having passed course Art, has a total of 5
credits. Student Bea has graduated in program Alpha.
```

Composing such texts demands little effort. For instance, once the record containing the event `enroll(Bea,Art)` is retrieved from the LOG, the first sentence is obtained by matching the template:

```
op_template(enroll(S,Co),
['Student ',S,' enrolled in course ',Co,'.']).
```

As a useful byproduct, these simple texts can be further processed towards the elaboration of *summary reports*, e.g. by *aggregating* [Deemter 2005] events into *conjunctive structures* (regardless of possible interspersed events) to form sentences like: `S1, S2, ..., and Sn enrolled in C`.

With predicate `show_plot`, the events are exhibited (figure 4) in *storyboard* format, with *comics* style images [Lima 2013]. The plot-dramatization program was written in C# and integrated with the SWI-Prolog environment through command-line arguments.

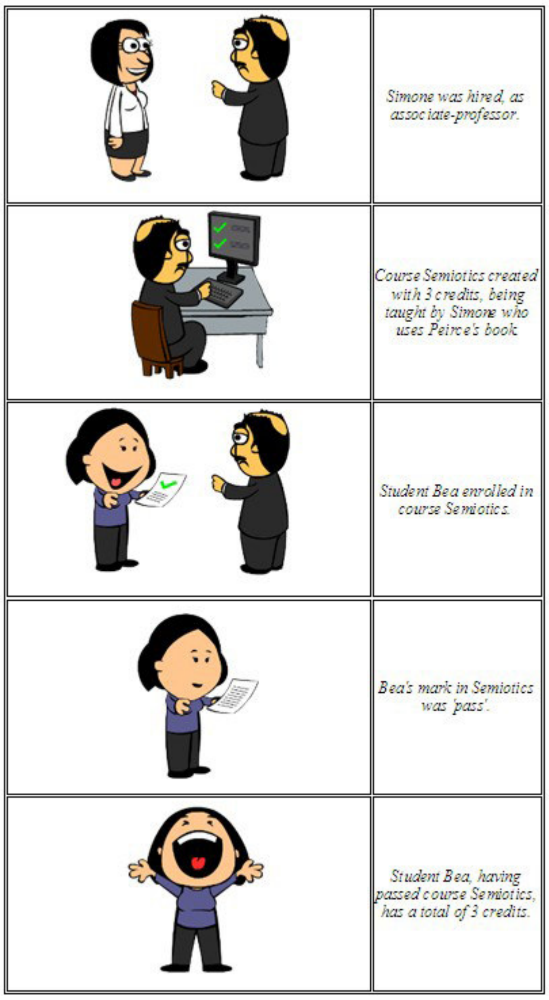

Simone was hired, as associate-professor.
Course Semiotics created with 3 credits, being taught by Simone who uses Peirce's book.
Student Bea enrolled in course Semiotics.
Bea's mark in Semiotics was 'pass'.
Student Bea, having passed course Semiotics, has a total of 3 credits.

Figure 4: Storyboard presentation

Clearly **Task 2** - *Monitoring the occurrence of situations* is most relevant, since watching for what may affect the agents' conduct stands out among the concerns one must have in mind. First in workspace and later in the database environments, the `sit_obj` rules (cf. section 3.3) ought to be tested for this purpose. One may eventually detect what follows, as a result of continuously monitoring the relational tables:

```
:- test_situations(student(S)).

<<Agent>>: student(Zoe)
<<Situation>>:
  student(Zoe)
  has_dropped(Zoe,Art)
  credits(Art,2)
  not takes(Zoe,A)
  course(Design)
  credits(Design,1)
  1<2
<<Objective>>:
  takes(Zoe,Design)
```

which indicates that the involved *agent*, student Zoe, faces a critical *situation*: she has dropped a two-credits course and is no longer enrolled in any course. In this regard an *objective* is recommended: start taking Design, presumably not so hard as Art, since it is merely (in our example!) no more than a one-credit course.

But suppose that when considering our second `sit_obj` rule, of interest to the Chairman, which looks for courses that have been offered for two years or more and no one has been able to finish successfully, this is found to be the case of the supposedly easy Design course:

```
:- test_situations(chairman).

<<Agent>>: chairman
<<Situation>>:
  course(Design)
  select_log(124, 2011/7/15/14/2/50/936,
             offer(Design,1))
  in_timestamp(year,2011,
              2011/7/15/14/2/50/936)
  in_current_time(year,2015)
  4 is 2015-2011
  4>=2
  not select_log(A,B,pass(C,Design,D,E))
<<Objective>>:
  not course(Design)
```

Once a situation that motivates a certain goal is detected, one should look for a suitable plan to satisfy the goal, which is the job of **Task 3** - *Choosing a plan to reach a goal*. The plan-generator may come up with a number of alternative plans, with different side effects. One criterion to choose the alternative to be executed is to take all such effects in consideration. Since a pre-condition to `cancel` a course is that there should be no student taking it, the planner, as expected, treats that as a preliminary sub-goal when responding to the Chairman's query:

```
?- goal_exec(not course('Design')).

Plan: start=>drop(Mary,Design)=>transfer(Joe,Design, Semiotics)=>cancel(Design)
with effects:
  not course(Design)
  not credits(Design,1)
  not takes(Mary,Design)
  not takes(Joe,Design)
  not grade(Mary,Design,pass)
  not grade(Joe,Design,inc)
  has_dropped(Mary,Design)
  has_dropped(Joe,Design)
  takes(Joe,Semiotics)
  grade(Joe,Semiotics,inc)
Want to execute this one? [yes/no/stop] - no

Plan: start=>drop(Mary,Design)=>mark(Joe,Design,pass)=>pass(Joe,Design,0,1)=>cancel(Design)
with effects:
  not course(Design)
  not credits(Design,1)
  not credits_won(Joe,0)
  not takes(Mary,Design)
  not takes(Joe,Design)
  not grade(Mary,Design,pass)
  not grade(Joe,Design,inc)
  credits_won(Joe,1)
  has_dropped(Mary,Design)
  has_finished(Joe,Design)
Want to execute this one? [yes/no/stop]
```

Again as expected, the firs plan employs `drop` and `transfer` as obvious ways to satisfy the pre-condition, but the second plan gives a strikingly different solution in Joe's case, much to the Chairman's surprise (and to ours, authors of the specification, who failed to recall – but the "system" would not – *that ceasing to take a course had been declared as one of the effects of the* `pass` *operation*!). Once this is revealed to be a possibility, it is up to those responsible for the policies regulating this particular system (the Chairman, in special) to decide whether or not it constitutes an acceptable alternative. In the negative case, a change in the specification (to be pursued all the way down to the implementation) may be in order.

We must note anyway, as we turn our attention to **Task 4** - *Finding patterns by comparing story sequences*, that the goals generated by the `sit_obj` rules are no more than *recommendations*, which the agents are not compelled to accept. Instead of behaving as rational plan-generation counsels, their conduct may follow recurring *patterns*, that can be extracted from the stories registered in the LOG, and that reflect the agents' common practice. Some of these patterns may indeed be regarded as *typical plans* to achieve goals that frequently arise, deserving to be kept as adaptable options in an easily accessible *library*.

As part of our project, we have been working on the construction of such libraries [Furtado 2001], employing *most specific generalization*, a method that constitutes the dual of *unification* [Minker 1988]. To derive a pattern, two or more sequences aiming at the same goal are first located, possibly in separate transactions, and then combined in a single sequence containing only the terms in common. Throughout the sequence the parameters of the terms are in turn generalized, so that constants that identically fill corresponding positions are kept, and variables are introduced consistently wherever different constants occur.

To attain even more generality, we adopt a different representation for plans, which we call a *plot data structure* [Karlsson 2009], in which the fact that plans may be no more than partially ordered is expressed by attaching identifying *tags* to each event and adding a dependence list, where a pair of tags f_i-f_j means that the event tagged with f_i should be executed before the one tagged f_j. For instance, enrolling a student in two courses can be done in any order, but a grade can only be assigned after the corresponding enrollment, as in the plan-plot conversion below:

```
plan:  start=>enroll(S,C1)=>enroll(S,C2)=>
       mark(S,C1,pass)
plot:  [[f1:enroll(S,C1),f2:enroll(S,C2),
         f3:mark(S,C1,pass),
        [f1-f3]]
```

Suppose, now, that Zoe, instead of enrolling in Design as recommended, prefers Semiotics (3 credits), presumably a more difficult course. Suppose further that the library contains a typical plan allowing those who have dropped 'Art' to still be able to graduate, even without meeting the 5 credits requirement of program Alpha. The solution shows that a "plan B" is available, leading to the appropriately named program Beta, with its more modest 4 credits requirement:

```
[[f1:drop(S,'Art'),
  f2:enroll(S,'Semiotics'),
  f3:enroll(S,'Design'),
  f4:mark(S,'Design',pass),
  f5:pass(S,'Design',0,1),
  f6:mark(S,'Semiotics',pass),
  f7:pass(S,'Semiotics',1,4),
  f8:receive_degree(S,'Beta')],
 [f1-f8,f2-f6,f3-f4,f4-f5,f5-f7,f6-f7,f7-f8]]
```

Matching the *observation* of Zoe's conduct against the above plot – a *plan-recognition* process – yields a successful unification, instantiating the `S` variable in the plot with the involved student's name:

```
:- Obs =
[drop('Zoe','Art'),enroll('Joe','Design')],
recogn(Obs,Plot,Found).

Found =
[[f1:drop(Zoe,Art),
  f2:enroll(Zoe,Semiotics),
  f3:enroll(Zoe,Design),
  f4:mark(Zoe,Design,pass),
  f5:pass(Zoe,Design,0,1),
  f6:mark(Zoe,Semiotics,pass),
  f7:pass(Zoe,Semiotics,1,4),
  f8:receive_degree(Zoe,Beta)],
 [f1-f8,f2-f6,f3-f4,f4-f5,f5-f7,f6-f7,f7-f8]]
```

which in fact does more than explaining Zoe's conduct: it allows us to predict that Zoe is expected, after all, to also enroll in Design sometime in the near future. This might be frustrated if the course is extinguished, as discussed earlier – we shall return to that shortly.

To compute statistics, the job of **Task 5** - *Applying statistical analysis to the* LOG, we employ the facilities offered by the R system, also accessible from SWI-Prolog. Figure 5 shows a bar diagram that compares the occurrences of event types.

This simple example aims at demonstrating how an administrator could leverage the LOG to query the system on *semantically relevant* information; as opposed to simple database usage statistics more commonly provided by DBMSs. Incidentally, the analysis of figure 5 allowed us to discover that, in this example domain, almost exclusively passing grades were being assigned to students, since the mark and pass events happen in the same proportion.

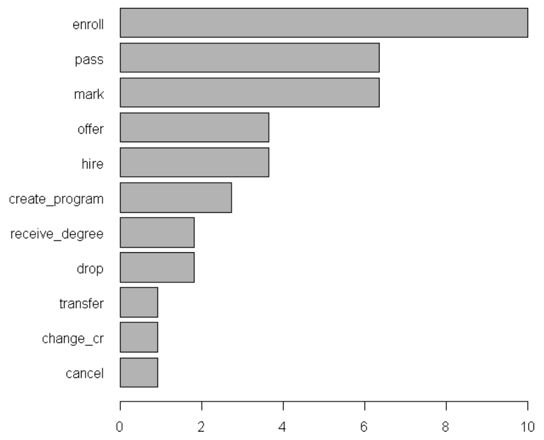


Figure 5: Proportional occurrence of events in the LOG

Coming to **Task 6** - *Retrieving past database states*, we realize that the presence of the LOG, together with the logic programming specification of the effects of the operations, contributes the functionalities of a *temporal database*. As a repository of past stories, the LOG contains the answer to temporal queries, such as: *"How many total credits had Joe in July 15th 2011, and what event produced that value?"*. To reply, one searches the LOG for an event, prior to the indicated date, that would be able to affect the value, and then makes sure that no event able to modify it has occurred between the timestamps of the two events.

```
:- holds_at(credits_won('Joe',T),2011/7/15,E).

T = 2,
E = pass(Joe,Art,0,2)
```

Moreover, it is possible to revert to a past state of the world, by, on the basis of the declarative specification of the operations, replicating the net effects of the events registered in the LOG since from the moment when the system started until the given date. Indicating as such the precise instant when Joe enrolled in Art, we reconstitute the entire set of facts then holding:

```
:- state_at(Fs,2011/07/15/14/03/48/061),
show_facts(Fs).

Course Art is being offered. Course Design is being offered. Course Semiotics is being
offered. Alpha is open as an academic program. Beta is open as an academic program. Joe is
registered as student. Course Art grants 2 credits. Course Design grants 1 credit. Course
Semiotics grants 3 credits. Program Alpha requires 5 credits. Program Beta requires 4
credits. Student Joe is taking course Art.
```

Yet, it must be admitted that, except when dealing with relatively small data repositories, the reconstitution of past database states by re-executing the operations in the LOG is unrealistic. Such overwhelming workload can hopefully be reduced to tolerable proportions by taking periodical snapshots of the database tables, so that the desired past states would be obtained, so-to-speak, by interpolation inside the time interval between two consecutive snapshots, for which only a manageable subsequence of the LOG need be processed.

With **Task 7** - *Scheduling future events*, we get a glimpse of the future. With the same column structure as the LOG, an AGENDA relational table was added to the **IDB** architecture, wherein events can be *scheduled* for possible future execution with a timestamp anticipating the intended date. If only the day is given, the current month and year are assumed; similarly, the current year is assumed if day and month are given explicitly. In any case, only future dates are acceptable. A serial number is added to the timestamp to establish a temporal order among events scheduled for the same day. The first parameter is the transaction's reference number. The example below should raise Zoe's expectations: the creation of a new course is envisaged:

```
?- schedule(130,offer('Logic',3,'Hermann',
   'Clocksin and Mellish'),10/10,7).
```

Like the LOG, the AGENDA can be consulted:

```
?- select_agenda(R,Ts,enroll('Zoe',C)).
R = 130,
Ts = 2015/10/10/7,
C = Logic .
```

and periodically inspected for optional execution of the items scheduled for the day (or pending, past due date). If execution is not solicited, the system asks whether the item should be deleted:

```
?- exec_agenda.

Want to execute <<enroll(Laura,Art)>>
reference 125, scheduled for 24 of September of 2015, item 1? [y/n]: n
Want to remove it from the Agenda? [y/n]: n
retained

Want to execute <<enroll(Zoe,Logic)>>
reference 128, scheduled for 1 of October of 2015, item 1? [y/n]: y
ok
executed
```

Pondering once again about Zoe and her unlucky colleagues who began with an ill-advised choice of course, the Chairman might ask the designers to provide some device that would allow students to find beforehand whether they would be apt to take a given course. One such device would be a multiple-choice test to verify if the candidate has bothered to acquire a minimum preliminary knowledge of the topic of the course. It was a simple matter to equip **IDB** with a test to be triggered for the problematic Art course:

```
?- course(C), eval_min('John',C,Result).

1. Who painted the Virgin of the Rocks?
a. Raphael               b. Picasso
c. Leonardo da Vinci     d. None of these
Your answer: c

2. Who composed the Magic Flute?
a. Mozart                b. Wagner
c. Puccini               d. None of these
Your answer: a

3. Who wrote the Divine Comedy?
a. Goethe                b. Dante
c. Victor Hugo           d. None of these
Your answer: b

4. Who sculpted the Thinker?
a. Phidias               b. Rodin
c. Michelangelo          d. None of these
Your answer: b

5. Who designed the Palácio da Alvorada?
a. Lucio Costa           b. Candido Portinari
c. Oscar Niemayer        d. None of these
Your answer: c

C = Art
Result = pass
```

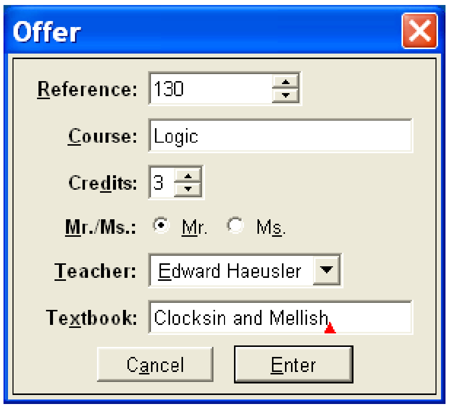
This little increment to the tool, as well as a few among the examples discussed in this section, concerning the tasks we chose to explore, illustrate another story level: the *meta-story* of the specification itself, whose inadequacies are revealed as the experiments proceed, mainly through plan generation, analysis of the LOG, and simulated execution. Application administrators and users have much to learn by watching how the system matures by trial and error, and how the mutually interfering rules that regulate it are more finely tuned so as to only enable stories with a happy ending (as far as the proprietary organization is concerned...). **IDB** employs a hosting system to operate this additional *version control* [Loeliger 2012] task.

6. Concluding remarks

The distinctive features of our project, namely conceptual modelling not only of facts but also of events and agents, the availability of a LOG to register past events, and of a plan generator and an AGENDA to project alternative futures, pave the way to a transition from *data*-bases to *story*-bases as a fundamental component of information systems.

Our approach eases the job of application administrators and of the various classes of prospective users to assess how effective is the specified system, by discovering in what kinds of stories they are invited to participate. Bringing to bear what we learned in another area whereon we have been similarly applying our plan-recognition/plan-generation paradigm, namely digital interactive storytelling [Ciarlini 2005], we endeavour to make the stories not only comprehensible but also somewhat more enjoyable, by telling them in colloquial natural language and playful images.

In the same vein, we are now actively investing on friendly end-user interfaces capable of encapsulating the logic programming interactions, such as those in the previous section. For instance, the Chairman can already, by filling-up the menu-driven dialog window of figure 6, produced via the XPCE package of SWI-Prolog, both register Logic (with a current faculty member) in the database (failure being signalled if the course already exists) and prepare an announcement.



Reference: 130
Course: Logic
Credits: 3
Mr./Ms.: [✓ Mr.] Ms.
Teacher: Edward Haeusler
Textbook: Clocksin and Mellish

```
Course Logic is now offered, with 3 credits,
taught by Mr. Edward Haeusler using Clocksin
and Mellish's book.
```

Figure 6: dialog window to offer and announce courses

Still concerning notation, it would be necessary to include facilities to process RDF triples if one wishes to incorporate data of Web provenance. Besides representing metadata about Web resources, RDF can also represent information about objects that can be identified on the Web, even when they cannot be directly retrieved from the Web [Breitman 2007]. That would seem to be a natural extension of our research, since, as noted by Chen [2002], "RDF can be viewed as a member of the Entity-Relationship model family".

Also, for scaling up the **IDB** prototype to cope with large business applications, more effort should be invested, both by enhancing the implemented algorithms and by adopting a modular divide-and-conquer design strategy [Casanova 1991, Graeffe 2014] to help reducing the size and complexity of the various tasks involved at each stage.

For future research, two lines seem particularly relevant. The first is to better arrange the environment for training purposes. Stories that emerge from the specification would be presented to the trainees with pauses between successive phases ("chapters"), as we already do in our **Logtell** project for the composition of fiction tales [Ciarlini 2005]. At each pause, they would be called to interact, by considering the current state of the mini-world and, using what they have learned so far about the domain in hand, by making decisions to influence how the (non-deterministic) story would branch. By transposing the prototype to a client-server architecture, multiuser participation would be enabled, after which a *game* feature might be incorporated to add attractiveness to training, with criteria to grade the participants while they compete and/or collaborate to reach goals, subject to scarce resource limitations.

The other line, that can be termed *plot-mining* or *story-mining*, looks even more promising, as evidenced by research in the field of *process mining* [Aalst 2011, Furtado 2007]. We believe that having a time-stamped LOG to register transactions – composed, as we have stressed here, of conceptually meaningful events – is a major asset towards a semantically and pragmatically richer approach to perform knowledge discovery over the processes that may occur during the lifetime of an information system.

References

Aalst, W. M. P. van der, 2011. Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer-Verlag, Berlin.

Barbosa, S. D. J. et al., 2015, Plot Generation with Character-Based Decisions. Computers in Entertainment, v. 12.

Barros, L. M., Musse, S. R., 2007. Planning algorithms for interactive storytelling. Computers in Entertainment, vol. 5, n 1. ACM, NY.

Batini, C., Ceri, S. & Navathe, S., 1991. *Conceptual Design: an Entity-Relationship Approach*. Addison-Wesley.

Breitman, K.K., Casanova, M.A., Truszkowski, W., 2007. Semantic Web: Concepts, Technologies and Applications. Springer.

Brewer, W. F., Lichtenstein, E. H., 1980. Event Schemas, Story Schemas and Story Grammars. Technical report No 197. University of Illinois at Urbana-Champaign.

Casanova, M.A. et al., 1991. A software tool for modular database design. ACM Transactions on Database Systems, vol. 16, no. 2, pp. 209-234.

Charniak, E., 1972. Toward a model of children's story comprehension. Technical Report AITR-266. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Chen, P.P. 2002. Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned. In *Software pioneers*. Springer.

Ciarlini, A. E. M., Furtado, A. L., 2002. Understanding and simulating narratives in the context of information systems. In *Proc. of 21st international conference on conceptual modeling*, Tampere, Finland.

Ciarlini, A.E.M., Pozzer, C.T., Furtado, A.L., Feijo, B., 2005. A Logic-Based Tool for Interactive Generation and Dramatization of Stories In: *International Conference on Advances in Computer Entertainment Technology (ACE 2005)*, Valencia.

Ciarlini, A.E.M. et al., 2010. Modeling Interactive Storytelling Genres as Application Domains. Journal of Intelligent Information Systems, v. 35, pp. 347-381.

Date, C.J., Darwen, H., Lorentzos, N.A., 2002. Temporal Data and the Relational Model. Morgan Kaufmann.

Deemter, K.V., Krahmer, E., Theune, M. , 2005. Real versus Template-Based Natural Language Generation: A False Opposition?. Computational. Linguistics, vol. 31, n. 1.

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).

Furtado, A.L., Casanova, M.A., Barbosa, S.D.J., Breitman, K.K., 2007. *Plot mining as an aid to characterization and planning*. Technical Report MCC07, PUC-Rio.

Furtado, A.L., Ciarlini, A.E.M., 2000. Generating Narratives from Plots using Schema Information. In: *Proc. of the 5th International Workshop on Applications of Natural Language for Information Systems*. Springer.

Furtado, A.L., Ciarlini, A.E.M., 2001. Constructing Libraries of Typical Plans. In: *Proc. of the 13th International Conference on Advanced Information Systems Engineering (CAiSE)*. Springer.

Graeffe, G. et al., 2014. In-memory performance for big data. In: *Proc. of the VLDB Endowment*, vol. 8, no 1.

Guttag J. 1977. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6).

Karlsson, B. et al., 2009. A plot-manipulation algebra to support digital storytelling. In: *Proc. of IFIP International Federation for Information Processing (ICEC)*.

Lang, R., 1999. A declarative model for simple narratives. In: *Proc. of the AAAI Fall Symposium on Narrative Intelligence*.

Lima, E.S., Feijó, B., Furtado, A.L., Barbosa, S.D.J., Pozzer, C.T., Ciarlini, A.E.M., 2013. Non-Branching Interactive Comics. In: *Proc. of the 10th International Conference on Advances in Computer Entertainment Technology*.

Loeliger, J., McCullough, M., 2012. Version Control with Git. O'Reilly Media.

Mateas, M. and Stern, A., 2003. Façade: An experiment in building a fully-realized interactive drama. Game Developers Conference.

Meehan, J., 1981. TALE-SPIN and Micro TALE-SPIN. In: Roger C. Schank, & Christopher K. Riesbeck (Eds.), Inside computer understanding. Hillsdale, NJ: Erlbaum.

Miller, R., Shanahan, M., 1994. Narratives in the Situation Calculus. Journal of Logic & Computation, Vol. 4., Number 5, 1994.

Minker, J., 1988. Foundations of Deductive Databases and Logical Programming. Morgan Kaufmann.

Prince, G., 1973. A Grammar of Stories. Mouton.

Propp, V., 1968. Morphology of the folktale. Austin, TX: University of Texas Press.

Riedl, M., Young, R. M., 2004. An intent-driven planner for multi-agent story generation. In: *Proc. of the 3rd International Conference on Autonomous Agents and Multi Agent Systems, July 2004.*

Rossi, F., Beek, P. 2006. Handbook of Constraint Programming, Elsevier Science.

Schank, R., Morson, G.S., 1995. Tell Me A Story. Northwestern University Press.

Zaniolo, C., 1992. Intelligent Databases: Old Challenges and New Opportunities. In: *Journal of Intelligent Information Systems* 1, 271-292.

Appendix

*****A Small Academic Example*****

% STATIC SCHEMA - classes of facts

```
entity(student,student_name).
attribute(student,credits_won).
entity(program,program_name).
attribute(program,requirement).
entity(course,course_name).
attribute(course,credits).
entity(teacher,teacher_name).
attribute(teacher,position).
relationship(takes,[student,course]).
attribute(takes,grade).
relationship(has_finished,[student,course]).
relationship(has_dropped,[student,course]).
relationship(graduated_in,[student,program]).
f_relationship(taught_by,[course,teacher]).
attribute(taught_by,textbook).
```

% DYNAMIC SCHEMA - operations to produce events

```
operation(offer(C,N,T,B)).
added(course(C),offer(C,N,T,B)).
added(credits(C,N),offer(C,N,T,B)).
added(taught_by(C,T),offer(C,N,T,B)).
added(textbook(C,T,B),offer(C,N,T,B)).
precond(offer(C,N,T,B),(/(not(course(C))),/teacher(T))).

operation(hire(T,P)).
added(teacher(T),hire(T,P)).
added(position(T,P),hire(T,P)).
precond(hire(T,P),true).

operation(enroll(S,C)).
/added(student(S),enroll(S,C)).
/added(credits_won(S,0),enroll(S,C)) :-
  not credits_won(S,_).
added(takes(S,C),enroll(S,C)).
added(grade(S,C,'inc'),enroll(S,C)).
precond(enroll(S,C),
  (course(C),
   /(not has_finished(S,C)),/(not has_dropped(S,C)))).

operation(drop(S,C)).
deleted(takes(S,C),drop(S,C)).
deleted(grade(S,C,G),drop(S,C)).
added(has_dropped(S,C),drop(S,C)).
precond(drop(S,C),true).

operation(transfer(S,C1,C2)).
deleted(takes(S,C1),transfer(S,C1,C2)).
deleted(grade(S,C1,G),transfer(S,C1,C2)).
added(has_dropped(S,C1),transfer(S,C1,C2)).
added(takes(S,C2),transfer(S,C1,C2)).
added(grade(S,C2,'inc'),transfer(S,C1,C2)).
```

```
precond(transfer(S,C1,C2),
  (course(C2),/takes(S,C1),
   /(not has_finished(S,C2)),/(not has_dropped(S,C2)))).

operation(cancel(C)).
deleted(course(C),cancel(C)).
deleted(credits(C,N),cancel(C)).
precond(cancel(C),
   not takes(S,C):(student(S),takes(S,C))).

operation(change_cr(C,N1,N2)).
deleted(credits(C,N1),change_cr(C,N1,N2)).
added(credits(C,N2),change_cr(C,N1,N2)).
precond(change_cr(C,N1,N2),credits(C,N1)).

operation(mark(S,C,G)).
deleted(grade(S,C,'inc'),mark(S,C,G)).
added(grade(S,C,G),mark(S,C,G)).
precond(mark(S,C,G),grade(S,C,'inc')).

operation(pass(S,C,T1,T2)).
deleted(credits_won(S,T1),pass(S,C,T1,T2)).
deleted(takes(S,C),pass(S,C,T1,T2)).
deleted(grade(S,C,G),pass(S,C,T1,T2)).
added(has_finished(S,C),pass(S,C,T1,T2)).
added(credits_won(S,T2),pass(S,C,T1,T2)).
precond(pass(S,C,T1,T2),
    (takes(S,C),credits_won(S,T1),grade(S,C,'pass'))) :-
      credits(C,N),
      T2 #= T1 + N, T1 #>= 0.

operation(create_program(P,R)).
added(program(P),create_program(P,R)).
added(requirement(P,R),create_program(P,R)).
precond(create_program(P,R),/(not program(P))).

operation(receive_degree(S,P)).
added(graduated_in(S,P),receive_degree(S,P)).
precond(receive_degree(S,P),
  (requirement(P,T),credits_won(S,T),/(not takes(S,_)))).
```

% BEHAVIOURAL SCHEMA - situation-objective rules

```
% a student who dropped a course and is not currently taking any course
% should enroll in some course with smaller number of credits, if such course is available

sit_obj(student(S),
        (  has_dropped(S,C1),
           credits(C1,Cr1),
           not takes(S,Cx),
           course(C2),
           credits(C2,Cr2),
           Cr2 < Cr1  ),
        takes(S,C2)).

% rule involving the LOG table accessed via ODBC
% courses created 2 or more years ago and not yet passed by any student
% should be cancelled by the chairman
```

```
sit_obj(chairman,
        (  course(C),
           select_log(R,Ts,offer(C,Cr)),
           in_timestamp(year,Yevent,Ts),
           in_current_time(year,Ynow),
           Diff is Ynow - Yevent, Diff >= 2,
           not select_log(_,_,pass(_,C,_,_))  ),
        not course(C)).
```

% GENERATED RELATIONAL TABLES

```
CREATE TABLE "STUDENT"
  ("STUDENT_NAME" VARCHAR2(100),
   "CREDITS_WON" NUMBER
  );
/

CREATE TABLE "PROGRAM"
  ("PROGRAM_NAME" VARCHAR2(100),
   "REQUIREMENT" NUMBER
  );
/

CREATE TABLE "COURSE"
  ("COURSE_NAME" VARCHAR2(100),
   "CREDITS" NUMBER,
   "TEACHER_NAME" VARCHAR2(100),
   "TEXTBOOK" VARCHAR2(100)
  );
/

CREATE TABLE "TEACHER"
  ("TEACHER_NAME" VARCHAR2(100),
   "POSITION" VARCHAR2(100)
  );
/

CREATE TABLE "TAKES"
  ("STUDENT_NAME" VARCHAR2(100),
   "COURSE_NAME" VARCHAR2(100),
   "GRADE" VARCHAR2(100)
  );
/

CREATE TABLE "HAS_FINISHED"
  ("STUDENT_NAME" VARCHAR2(100),
   "COURSE_NAME" VARCHAR2(100)
  );
/

CREATE TABLE "HAS_DROPPED"
  ("STUDENT_NAME" VARCHAR2(100),
   "COURSE_NAME" VARCHAR2(100)
  );
/

CREATE TABLE "GRADUATED_IN"
```

```
  ("STUDENT_NAME" VARCHAR2(100),
   "PROGRAM_NAME" VARCHAR2(100)
  );
/

CREATE TABLE "LOG"
  ("REF" NUMBER,
   "TS" VARCHAR2(100),
   "EVENT" VARCHAR2(100)
   );
/

CREATE TABLE "AGENDA"
  ("REF" NUMBER,
   "TS" VARCHAR2(100),
   "EVENT" VARCHAR2(100)
   );
/

CREATE TABLE "REF"
  ("R" NUMBER
   );
/
```

% COMPILED OPERATIONS IN PROCEDURAL-STYLE

```
offer(A, C, B, D) :-
        ref(E),
        not select(course(A)),
        select(teacher(B)),
        insert(course(A, C, B, D)),
        ins_log(E, offer(A, C, B, D)).

hire(A, B) :-
        ref(C),
        insert(teacher(A, B)),
        ins_log(C, hire(A, B)).

enroll(B, A) :-
        ref(C),
        select(course(A)),
        not select(has_finished(B, A)),
        not select(has_dropped(B, A)),
        (   select(student(B))
        ;   not select(student(B)),
            insert(student(B, 0))
        ),
        insert(takes(B, A, inc)),
        ins_log(C, enroll(B, A)).

drop(A, B) :-
        ref(C),
        delete(takes(A, B, _)),
        insert(has_dropped(A, B)),
        ins_log(C, drop(A, B)).

transfer(B, C, A) :-
        ref(D),
        select(course(A)),
        select(takes(B, C)),
        not select(has_finished(B, A)),
        not select(has_dropped(B, A)),
        delete(takes(B, C, _)),
        insert(has_dropped(B, C)),
        insert(takes(B, A, inc)),
        ins_log(D, transfer(B, C, A)).

cancel(A) :-
        ref(B),
        not select(takes(_, A)),
        delete(course(A, _, _, _)),
        ins_log(B, cancel(A)).

change_cr(A, B, C) :-
        ref(D),
        select(credits(A, B)),
        update(course(A, credits: (B=>C))),
        ins_log(D, change_cr(A, B, C)).

mark(A, B, C) :-
        ref(D),
        select(grade(A, B, inc)),
        update(takes((A, B), grade: (inc=>C))),
        ins_log(D, mark(A, B, C)).

pass(A, B, C, D) :-
        ref(F),
        select(takes(A, B)),
        select(credits_won(A, C)),
        select(grade(A, B, pass)),
        select(credits(B, E)),
        D#=C+E,
        delete(takes(A, B, _)),
        insert(has_finished(A, B)),
        update(student(A, credits_won: (C=>D))),
        ins_log(F, pass(A, B, C, D)).

create_program(A, B) :-
        ref(C),
        not select(program(A)),
        insert(program(A, B)),
        ins_log(C, create_program(A, B)).

receive_degree(B, A) :-
        ref(D),
        select(requirement(A, C)),
        select(credits_won(B, C)),
        not select(takes(B, _)),
        insert(graduated_in(B, A)),
        ins_log(D, receive_degree(B, A)).
```

% COMPILED STORED PROCEDURES

```
create procedure offer(rn number, A varchar2, B number, C varchar2, D varchar2) is
 found number;
 ev varchar(100);
 teacher_name_C varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select count(*) into found
 from course
 where course_name = A;
 if found > 0 then return;
 end if;
 select teacher_name into teacher_name_C
 from teacher
 where teacher_name = C;
 insert into course
 values (A, B, C, D);
 ev := 'offer(' || A || ',' || B || ',' || C || ',' || D || ')';
 ins_log(rn,ev);
 commit;
end offer;
/

create procedure hire(rn number, A varchar2, B varchar2) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 insert into teacher
 values (A, B);
 ev := 'hire(' || A || ',' || B || ')';
 ins_log(rn,ev);
 commit;
end hire;
/

create procedure enroll(rn number, A varchar2, B varchar2) is
 found number;
 ev varchar(100);
 course_name_B varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select course_name into course_name_B
 from course
 where course_name = B;
 select count(*) into found
 from has_finished
 where student_name = A and course_name = B;
 if found > 0 then return;
 end if;
 select count(*) into found
 from has_dropped
 where student_name = A and course_name = B;
 if found > 0 then return;
 end if;
 select count(*) into found
 from student
 where student_name = A;
 if found = 0 then
 insert into student
 values (A, 0);
 end if;
 insert into takes
 values (A, B, 'inc');
 ev := 'enroll(' || A || ',' || B || ')';
 ins_log(rn,ev);
 commit;
end enroll;
/

create procedure drop(rn number, A varchar2, B varchar2) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 delete from takes
 where student_name = A and course_name = B;
 insert into has_dropped
 values (A, B);
 ev := 'drop(' || A || ',' || B || ')';
 ins_log(rn,ev);
 commit;
end drop;
/

create procedure transfer(rn number, A varchar2, B varchar2, C varchar2) is
 found number;
 ev varchar(100);
 course_name_B varchar(100);
 course_name_C varchar(100);
 student_name_A varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select course_name into course_name_C
 from course
 where course_name = C;
 select student_name,course_name into student_name_A,course_name_B
 from takes
 where student_name = A and course_name = B;
 select count(*) into found
 from has_finished
 where student_name = A and course_name = C;
 if found > 0 then return;
 end if;
 select count(*) into found
 from has_dropped
 where student_name = A and course_name = C;
 if found > 0 then return;
 end if;
 delete from takes
 where student_name = A and course_name = B;
 insert into has_dropped
 values (A, B);
 insert into takes
 values (A, C, 'inc');
 ev := 'transfer(' || A || ',' || B || ',' || C || ')';
 ins_log(rn,ev);
 commit;
end transfer;
/

create procedure cancel(rn number, A varchar2) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select count(*) into found
 from takes
 where course_name = A;
 if found > 0 then return;
 end if;
 delete from course
 where course_name = A;
 ev := 'cancel(' || A || ')';
 ins_log(rn,ev);
 commit;
end cancel;
/

create procedure change_cr(rn number, A varchar2, B number, C number) is
 found number;
 ev varchar(100);
 course_name_A varchar(100);
 credits_B number;
begin
 select r into found
 from ref
 where r = rn;
 select course_name,credits into course_name_A,credits_B
 from course
 where course_name = A and credits = B;
 update course
 set credits=C
 where course_name=A;
 ev := 'change_cr(' || A || ',' || B || ',' || C || ')';
 ins_log(rn,ev);
 commit;
end change_cr;
/

create procedure mark(rn number, s varchar2, c varchar2, g varchar2) is
 found number;
 ev varchar(100);
 course_name_c varchar(100);
 grade_inc varchar(100);
 student_name_s varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select student_name,course_name,grade into student_name_s,course_name_c,grade_inc
 from takes
 where student_name = s and course_name = c and grade = 'inc';
 update takes
 set grade=g
 where student_name=s and course_name=c;
 ev := 'mark(' || s || ',' || c || ',' || g || ')';
 ins_log(rn,ev);
 commit;
end mark;
/

create procedure pass(rn number, A varchar2, B varchar2, C number, D in out number) is
 found number;
 ev varchar(100);
 course_name_B varchar(100);
 credits_E number;
 credits_won_C number;
 grade_pass varchar(100);
 student_name_A varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select student_name,course_name into student_name_A,course_name_B
 from takes
 where student_name = A and course_name = B;
 select student_name,credits_won into student_name_A,credits_won_C
 from student
 where student_name = A and credits_won = C;
 select student_name,course_name,grade into student_name_A,course_name_B,grade_pass
 from takes
 where student_name = A and course_name = B and grade = 'pass';
 select course_name,credits into course_name_B,credits_E
 from course
 where course_name = B;
 D:=C+credits_E;
 delete from takes
 where student_name = A and course_name = B;
 insert into has_finished
 values (A, B);
 update student
 set credits_won=D
 where student_name=A;
 ev := 'pass(' || A || ',' || B || ',' || C || ',' || D || ')';
 ins_log(rn,ev);
 commit;
end pass;
/

create procedure create_program(rn number, A varchar2, B number) is
 found number;
 ev varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select count(*) into found
 from program
 where program_name = A;
 if found > 0 then return;
 end if;
 insert into program
 values (A, B);
 ev := 'create_program(' || A || ',' || B || ')';
 ins_log(rn,ev);
 commit;
end create_program;
/

create procedure receive_degree(rn number, A varchar2, B varchar2) is
 found number;
 ev varchar(100);
 credits_won_C number;
 program_name_B varchar(100);
 requirement_C number;
 student_name_A varchar(100);
begin
 select r into found
 from ref
 where r = rn;
 select program_name,requirement into program_name_B,requirement_C
 from program
 where program_name = B;
 select student_name,credits_won into student_name_A,credits_won_C
 from student
 where student_name = A and credits_won = requirement_C;
 select count(*) into found
 from takes
 where student_name = A;
 if found > 0 then return;
 end if;
 insert into graduated_in
 values (A, B);
 ev := 'receive_degree(' || A || ',' || B || ')';
 ins_log(rn,ev);
 commit;
end receive_degree;
/

create procedure ins_log(ref number, ev varchar2) is
ts varchar(100);
begin
select to_char(localtimestamp,'YYYY/MM/DD/HH24/MI/SS/FF3')
into ts
from DUAL;
insert into log
values (ref,ts,ev);
commit;
end ins_log;
/
```