

Analysis and Reuse of Plots using Similarity and Analogy

Antonio L. Furtado, Marco A. Casanova,
Simone D.J. Barbosa, Karin K. Breitman

Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de S. Vicente, 225 – Rio de Janeiro, Brasil – CEP 22451-900
{furtado, casanova, simone, karin}@inf.puc-rio.br

Abstract: A plot is a partially ordered set of events. Plot analysis is a relevant source of knowledge about the agents' behavior when accessing data stored in the database. It relies on logical logs which register the actions of individual agents. This paper proposes techniques to analyze and reuse plots based on the concepts of similarity and analogy, borrowed from cognitive science and linguistics. The concept of similarity is applied to organize plots as a library, and to explore the reuse of plots in the same domain. By contrast, the concept of analogy helps reuse plots across different domains. The techniques proposed in this paper find applications in areas such as computer games and emergency response information systems, as well as some traditional business applications.

Keywords: Temporal database, log, plot, narrative, similarity, analogy.

1. Introduction

Literary research addresses narratives at successive levels. The most basic level, the *fabula*, is defined as "a series of logically and chronologically related events that are caused or experienced by actors" [3]. A series of events is often called a *plot*, a notion that can be usefully transposed to the context of information systems. Intuitively, plots are the stories [25,26] that happen in the underlying mini-world and, as a result, produce state-changes in its database representation. More precisely, an event represents the result of the execution of some domain-oriented operation by an authorized agent, and a plot is a partially ordered set of events. What logically relates the events, and determines the precedence among them, is the interplay between the pre- and post-conditions in terms of which operations are defined. The pre- and post-conditions, in turn, reflect the integrity constraints and business rules prevailing in the application domain involved.

Plot analysis is a rich source of knowledge about the agents' behavior when accessing data stored in the database. It relies on (logical) database logs, also called audit trails, which register the actions of individual agents. A trivial example of a log is a bank account statement, which records the sequence of actions executed against the account. A second example comes from storytelling engines, such as **LOGTELL** [9,10], which model the world as a database and are based on a set of pre-defined

actions and plots [10]. A log in this case is the trace of events generated by composing a story interactively. In the context of an emergency response information system [27], a log registers the actions taken when handling an emergency, or during a training exercise [8].

The thrust of this paper is to propose techniques to analyze and reuse plots applying the concepts of similarity and analogy, borrowed from cognitive science and linguistics [5,18]. We used these notions to explore database conceptual design techniques and query interfaces at previous stages of our research project [7,4]. In the present paper, we first discuss how to extract plots from logs. Then, we explore the concept of similarity to organize a plot library that helps reuse plots in the same domain. By contrast, we apply the concept of analogy to reuse plots across *different* domains.

Returning to our examples, we sometimes spend countless hours analyzing our bank account statement (our account log) to figure out recurrent similar groups (similar plots) of funds transfers, deposits, withdrawals and payments (the events in banking applications). Plots in this case are typically simple, say, a set of withdrawals and payments followed by a deposit or funds transfer to balance the account. For example, we may be overspending on certain weekends, which come after the (forgotten) due dates of a number of bills. If similar groups of events repeat month after month, i.e., similar plots reoccur every month, then we ought to change our spending pattern. Usually, such groups of events are not treated as plots because the relationships between the events are too simple: the withdrawals and payments merely precede the deposit or funds transfer. One may therefore use traditional data mining techniques to detect the pattern of recurring events.

As a second example, from the domain of digital storytelling, consider the **LOGTELL** engine which features a built-in planner that creates new plots from a library of pre-defined operations and plots. By analyzing plot similarities, the game designer may come out with recurrent patterns that he may reuse to generate new plots in other domains, by analogy. An entirely compatible approach has been proposed a long time ago in literary theory by the Russian researcher Vladimir Propp [24]. In order to specify the genre of fairy-tales, he described a set of 31 functions, comparable to what we are calling domain-oriented, or more appropriately here, genre-oriented operations, which he claimed to be enough to account for a large sample extracted from an anthology of fairy-tales compiled by Aleksandr Afanas'ev [1]. Propp's research in fact focused on finding analogous plots in different fairy-tales.

The third example comes from the domain of emergency response information systems, where logs are an essential resource [8]. A plot in this case is a set of interrelated actions that an emergency team must perform to mitigate a specific accident scenario during an emergency, such as to isolate and clean an area affected by the spill of a hazardous material. An analysis of the log of actions taken during the response to an accident may help settle legal disputes. Furthermore, if plots generated as responses to similar accident scenarios exhibit similar inappropriate sequences of mitigating actions, then the analysis is an indication that the emergency teams, equipment or procedures must be revised. Similarly, emergency response information systems are valuable tools to train emergency teams on how to react to accidents. In

this case, the logs will be the result of simulated accidents. An analysis of the plots extracted from the logs will help assess how prepared the teams are, or else if some procedure is not appropriate, which is detected when the teams repeatedly generate similar inappropriate sequences of mitigating actions as a response to similar accidents. Plot analogy may also be a valuable pedagogical strategy to train the teams, i.e., if a team is trained in one type of accident, one might try to transpose, by analogy, the scenario and the plot mastered by the team to a different scenario and a different group of interrelated actions.

The practical relevance of the concepts of similarity and analogy, central to our work, has been amply recognized. Winston [29] is an early reference that describes a theory of analogy with applications to AI systems. Metaphors [18] have been used to improve human-computer interface design [5], in particular, and software design, in general [20]. The use of similarity and analogy in the context of database design has been explored in [7,4]. The plot extraction technique we adopt follows the AI tradition [15]. Our approach differs from research, such as [21], which focuses on the mining of rules from sequence databases, which are mostly based on statistical confidence levels. The notion of plot similarity we adopt also differs from the notion of process similarity introduced in [2], which computes a coefficient that indicates how dissimilar two processes are. Aalst et al. [28] describe an algorithm to extract a process model from such a log and represent it in terms of a Petri net. Their approach differs from ours in so far as the plot extraction technique, described in Section 2.3, is based on the semantics of the operations, expressed by pre- and post-conditions.

The paper is organized as follows. Section 2 introduces plots and related concepts, and outlines an algorithm to extract plots from logs. Section 3 discusses how to reuse plots in the same domain, employing the concept of similarity. Section 4 considers the reuse of plots across different domains, resorting to the notion of analogy. Section 5 contains the conclusion.

2. Basic concepts and techniques

2.1 Informal characterization of the basic concepts

In this section, we informally introduce the basic concepts we use throughout the paper. Our notation is based on Prolog, from which we borrow variables, constants and literals [7,11]. Section 2.2 contains rigorous definitions for the main concepts.

Our database conceptual schema follows the usual conventions of the Entity-Relationship model [6]. In addition, the schema contains a repertoire of domain-oriented operations, defined through their pre- and post-conditions, as proposed in the STRIPS system [12]. In order to preserve the integrity constraints regulating the mini-world represented in the database, an operation can execute only if its pre-conditions currently hold, and the effect of its execution corresponds precisely to its post-conditions. A complementary requirement is that the state of the database may change only by executing an operation from the predefined repertoire.

To illustrate these concepts, we introduce a simple example schema in the domain of products and components:

4 Antonio L. Furtado, Marco A. Casanova,
Simone D.J. Barbosa, Karin K. Breitman

```

entity classes: product, component
attributes:    pno of product
              cno, ctype, defective of component
relationship: iscompof associating component with product
operations:   repair(pno, cno)
              order(ctype, cno)
              replace(pno, cno1, cno2)

```

where instances of `product` are identified by `pno`, and instances of `component`, which is a *weak entity* [6], are identified via the `iscompof` relationship combined with the discriminating attribute `cno`. The value of attribute `ctype` indicates the type of a component, and the Boolean attribute `defective` exclusively qualifies those components that have been found to be defective.

Using the Prolog-based notation introduced in [7], the specification of the Product schema would be:

```

Schema: Product
Clauses --
entity(product, pno)
attribute(product, pno)
entity(component, [pno/cno-iscompof-pno, cno])
attribute(component, cno)
attribute(component, ctype)
attribute(component, defective)
relationship(iscompof, component/n/total, product/1/total)
operation(order, [ctype, cno])
pre(order(A, B), [])
post(order(A, B), [ctype(B, A), -defective(B)])
operation(replace, [pno, cno, cno])
pre(replace(A, B, C),
    [iscompof(B, A), ctype(B, D), ctype(C, D)]/diff(B, C)
post(replace(A, B, C),
    [-iscompof(B, A), iscompof(C,A)]/diff(B, C)
operation(repair, [pno, cno])
pre(repair(A, B), [defective(B)])
post(repair(A, B), [-defective(B)])

```

A database conceptual schema may also include *goal-inference rules* of the form $S \rightarrow G$, where S is a *situation* and G a *goal*, both of which are sets of literals. Such rules capture the motivation of an agent who, observing that a certain situation S holds, would be expected to execute the appropriate operations to reach a state where the goal G holds.

The conceptual schema we just introduced may include, for example, the goal-inference rule $S \rightarrow G$, where

```

S = [iscompof(X, Y), ctype(X, Z), defective(X)]
G = [iscompof(W, Y), ctype(W, Z), -defective(W)]

```

To conclude, we define events, logs and plots. An *event* is a statement of the form $o(p_1, \dots, p_n)$, where o is an operation name and (p_1, \dots, p_n) is the parameter list of the operation, which is a list of (Prolog) terms. An event $o(p_1, \dots, p_n)$ is *ground* iff the parameter list does not contain variables.

A *log* is a sequence of ground events.

A *plot* is a pair $P=(E,D)$, where E is a set of events and $D \subseteq E \times E$ is a partial order over E . A plot $P=(E,D)$ is *ground* iff all events in E are ground.

The relation D defines a set of *precedence dependencies* over E and captures the idea that, if an event e_1 contributes, as a result of its post-conditions, to the pre-conditions of another event e_2 , then e_1 must precede e_2 in the plot.

The precedence dependencies in a plot are denoted with the help of *tags*. More precisely, a plot is denoted as two lists: a list of expressions of the form $t : e$, where t is a constant, called a *tag*, and e is an event; and a list of expressions of the form $t - u$, where t and u are tags used in the first list. Tags are just a notational convenience, so that the plots P_1 and P_2 below are treated as *equivalent* plots:

$$\begin{aligned} P_1 &= [[f1: \text{order}(ct, c4), f2: \text{replace}(pr, c3, c4)], [f1-f2]] \\ P_2 &= [[f7: \text{replace}(pr, c3, c4), f8: \text{order}(ct, c4)], [f8-f7]] \end{aligned}$$

As will be argued in the next sections, it is in general a useful practice to record a plot P together with an associated goal-inference rule. We therefore define an *indexed plot* as a triple (S,G,P) , where P is a plot and $S \rightarrow G$ is a situation-goal rule, called the *circumstance* associated with P .

2.2 Formal characterization of the basic concepts

In this section, we define the syntax and semantics of the basic concepts that support the Prolog implementation of the plot techniques described in this paper. For brevity, we use standard concepts from first-order languages without definition. We refer the reader to [10] for the details.

A *static ER language* \mathcal{E} is a many-sorted first-order language whose alphabet contains a set \mathcal{D} of *database symbols* to describe database conceptual objects.

A *substitution* is a function θ that maps variables of \mathcal{E} into terms of \mathcal{E} (of the same sort). A substitution θ is *ground* iff it maps variables of \mathcal{E} into variable-free terms of \mathcal{E} . A substitution is *total* iff it is defined for all variables of \mathcal{E} ; otherwise, it is *partial*. If otherwise indicated, we assume that a substitution is total.

An expression ϕ is a formula or a term of \mathcal{E} . Given a substitution θ , we use $\phi\theta$ to denote the expression obtained by applying θ to ϕ . A (*ground*) *instance* of ϕ is an expression obtained by applying a (*ground*) substitution to ϕ .

A *literal* is an expression of the form $p(t_1, \dots, t_n)$ or of the form $\neg p(t_1, \dots, t_n)$, where p is an n -ary predicate symbol of \mathcal{E} and t_1, \dots, t_n is a list of terms of \mathcal{E} . A *database literal* is a literal whose predicate symbol is in \mathcal{D} , and a *database fact* is a ground positive database literal.

A *structure* M for \mathcal{E} assigns to each symbol s of \mathcal{E} an *interpretation* s^M as for first-order languages. The notion that M *satisfies* a formula of \mathcal{E} is also defined as usual. A *possible fact* of M is a database fact of \mathcal{E} that is satisfied by M . A *possible database state* of M is a set of possible facts of M .

Let G be a conjunction (or a set) of ground database literals, K be a conjunction (or a set) of database literals, and s be a possible database state. Then,

- s *satisfies* G for M , denoted $s \models_M G$, iff $g \in s$, for each ground literal g that occurs in G

- s satisfies K for M , denoted $s \models_M K$, iff $s \models_M G$, for each ground instance G of K

A *static ER schema* is a pair $\mathcal{S}=(\mathcal{E},\mathcal{O})$ such that \mathcal{E} is a static ER language and \mathcal{O} is a set of formulas of \mathcal{E} , called the *axioms* of \mathcal{S} , that includes *ER constraints*, which capture ER concepts, and *domain constraints*, which capture properties of the application domain. A *model* of \mathcal{S} is a structure of \mathcal{E} that satisfies all axioms in \mathcal{O} .

A *dynamic ER language* \mathcal{L} is a static ER language whose alphabet is extended to include a new set of symbols, the *operation names*, with an associated *arity*, and whose set of expressions is extended to include operation specifications, events, and situation-goal rules.

An *operation specification* for an n -ary operation name o is an expression \mathcal{O} of the form $\{P\}o(x_1, \dots, x_n)\{Q\}$, where x_1, \dots, x_n is a list of distinct variables, and P and Q are sets of database literals. We say that $o(x_1, \dots, x_n)$ is the *input declaration*, P is the *pre-condition* and Q is the *post-condition* of \mathcal{O} .

A structure M for \mathcal{L} is defined as for static ER languages, except that M assigns to each operation name o a set o^M of pairs of possible database states of M .

Let \mathcal{O} be an operation specification for an n -ary operation name o and assume that \mathcal{O} is of the form $\{P\}o(x_1, \dots, x_n)\{Q\}$. Let θ be a ground substitution of \mathcal{L} . Then, a pair (s, t) of possible database states in o^M satisfies $\mathcal{O}\theta$ for M iff

- $s \models_M P\theta$ (the pre-conditions are satisfied in s for M)
- $t \models_M Q\theta$ (the post-conditions are satisfied in t for M)
- for every possible database fact f of M , if neither f nor $\neg f$ occur in $Q\theta$, then $t \models f$ iff $s \models f$ (which is the frame requirement: preservation of satisfaction from s to t for ground database literals that are neither established nor negated by the post-condition $Q\theta$, which is ground by assumption)

Furthermore, M satisfies $\mathcal{O}\theta$ iff every pair (s, t) of possible database states in o^M satisfies $\mathcal{O}\theta$ for M . Finally, M satisfies \mathcal{O} iff M satisfies every ground instance of \mathcal{O} .

An *event* is an expression e of the form $o(t_1, \dots, t_n)$, where o is an n -ary operation name of \mathcal{L} and t_1, \dots, t_n is a list of terms of \mathcal{L} . The *parameter substitution* of e is the partial substitution θ_e that maps x_i into t_i , for $i \in [1, n]$, and is undefined for the other variables of \mathcal{L} . We define an interpretation in M for e as the set e^M consisting of all pairs (s, t) of possible database states in o^M such that (s, t) satisfies $\mathcal{O}\rho$ for M , where $\rho = \theta_e \circ \varphi$, for some ground substitution φ .

Let e be an event of the form $o(t_1, \dots, t_n)$, \mathcal{O} be the operation specification for o , and θ_e be the parameter substitution of e . Then, $\mathcal{O}\theta_e$ is the *specification for e induced by \mathcal{O}* . A structure M satisfies $\mathcal{O}\theta_e$ with respect to \mathcal{O} iff M satisfies every ground instance $\mathcal{O}\rho$ of \mathcal{O} , where $\rho = \theta_e \circ \varphi$, for some ground substitution φ .

A *temporal database* of a structure M is a sequence $\mathcal{S}=(s_0, s_1, \dots)$ of possible database states of M .

A *situation-goal rule* is an expression of the form $S \rightarrow G$, where S and G are sets of database literals. A temporal database $\mathcal{S}=(s_0, s_1, \dots)$ of M *satisfies* $S \rightarrow G$, denoted $\mathcal{S} \models_M S \rightarrow G$, iff there are p and q , with $1 \leq p \leq q \leq |\mathcal{S}|$, such that $s_p \models_M S$ and $s_q \models_M G$.

A *dynamic ER schema* is a pair $\mathcal{D}=(\mathcal{L}, \mathcal{A})$ such that \mathcal{L} is a dynamic ER language and \mathcal{A} is a set of formulas of \mathcal{L} , called the *axioms* of \mathcal{T} , that includes ER constraints, domain constraints, operation specifications and situation-goal rules, such that, for each operation name o of \mathcal{L} , there is exactly one operation specification for o in \mathcal{A} . A *model* of \mathcal{D} is a structure for \mathcal{L} that satisfies all axioms in \mathcal{A} .

Let e be an event of \mathcal{L} and assume that e is of the form $o(t_1, \dots, t_n)$. One can prove that, if M is a model of \mathcal{D} , then M satisfies $\theta\theta_e$ with respect to the (unique) operation specification for o that occurs in \mathcal{A} .

We prefer to introduce logs and plots as meta-level concepts, rather than expressions of dynamic ER languages, to avoid complex syntactical structures. Let \mathcal{L} be a dynamic ER language and M be a structure of \mathcal{L} in what follows.

A *log* is a possibly empty finite sequence $\mathbf{E}=(e_1, e_2, \dots, e_n)$ of ground events of \mathcal{L} . A temporal database $\mathcal{S}=(s_0, s_1, \dots)$ of M *satisfies* $\mathbf{E}=(e_1, e_2, \dots, e_n)$, denoted $\mathcal{S} \models_M \mathbf{E}$, iff $|\mathcal{S}| > n$ and, for each $i \in [1, n]$, $(s_{i-1}, s_i) \in e_i^M$ (i.e. e_i caused the transition from s_{i-1} to s_i).

A *plot* is a pair $P=(P_E, P_D)$, where P_E is a finite set of events and $P_D \subseteq P_E \times P_E$ is a partial order over P_E . A log $\mathbf{E}=(e_1, e_2, \dots, e_n)$ is *consistent with* $P=(P_E, P_D)$ iff there is a ground substitution θ such that, for each event e in P_E , the ground instance $e\theta$ of e occurs in \mathbf{E} and, for every $e_1, e_2 \in P_E$, if $(e_1, e_2) \in P_D$, then $e_1\theta$ precedes $e_2\theta$ in \mathbf{E} .

A temporal database $\mathcal{S}=(s_0, s_1, \dots)$ of M *satisfies* $P=(P_E, P_D)$, denoted $\mathcal{S} \models_M P$, iff there is a log $\mathbf{E}=(e_1, e_2, \dots, e_n)$ such that \mathcal{S} satisfies \mathbf{E} and \mathbf{E} is consistent with P . Finally, a plot P is *consistent* with a set Π of situation-goal rules with respect to a structure M iff, for every temporal database \mathcal{S} of M , if \mathcal{S} satisfies P then \mathcal{S} also satisfies Π .

2.3 Extracting indexed plots from a log

In this section, we briefly introduce an algorithm to extract indexed plots from a log. The details of the algorithm can be found in [13].

The algorithm takes as input a goal-inference rule $S \rightarrow G$ and a log L , and outputs an indexed plot (S_P, G_P, P) . The first step of the algorithm uses a simulation process that essentially recapitulates the evolution of the database while traversing the log. A sub-sequence M is extracted from the log L iff, prior to the execution of the first event in M , the situation S holds and, after the execution of the last event in M , a state is reached where G holds. This process may generate ground instances S_P and G_P of S and G , respectively. A plot P is obtained from M by a filtering process that keeps only the events whose post-conditions contribute to G_P and in addition, proceeding backwards recursively, those events that contribute to pre-conditions of events already included in P . The algorithm then outputs the indexed plot (S_P, G_P, P) .

For example, consider the goal-inference rule $S \rightarrow G$, where

$$\begin{aligned} S &= [\text{iscompof}(X, Y), \text{ctype}(X, Z), \text{defective}(X)] \\ G &= [\text{iscompof}(W, Y), \text{ctype}(W, Z), \neg\text{defective}(W)] \end{aligned}$$

and suppose that S and G are found to hold, respectively, before the first event, and immediately after the last event of the sub-sequence of the log shown below:

... order(ct, c4) ... replace(pr, c3, c4) ...

In view of the pre- and post-conditions of `order` and `replace`, defined in Section 2.1, the algorithm then outputs the indexed plot (S_p, G_p, P) , where

$$\begin{aligned} S_p &= [\text{iscompof}(c3, \text{pr}), \text{ctype}(c3, \text{ct}), \text{defective}(c3)] \\ G_p &= [\text{iscompof}(c4, \text{pr}), \text{ctype}(c4, \text{ct}), \neg\text{defective}(c4)] \\ P &= [[f1: \text{order}(\text{ct}, c4), f2: \text{replace}(\text{pr}, c3, c4)], [f1-f2]] \end{aligned}$$

3. Using similarity to organize and reuse plots

3.1 The notion of plot and indexed plot similarity

A *similarity mapping* ρ is a bijective mapping between terms, that is, a set of pairs (u, v) of terms such that any two pairs are equal on the first component iff they are equal on the second component. Let K be a set of literals, e be an event, $P=(E, D)$ be a plot, and κ be a situation-goal rule of the form $S \rightarrow G$. Then, the expression $K\rho$ denotes the set $\{l\rho \mid l \in K\}$, the expression $e\rho$ denotes the event obtained by replacing each term u that occurs in e by v , if $(u, v) \in \rho$, the expression $P\rho$ denotes the plot (F, G) such that $F = \{e\rho \mid e \in E\}$ and $G = \{(e, f)\rho \mid (e, f) \in D\}$, and the expression $\kappa\rho$ denotes the situation-goal rule $S\rho \rightarrow G\rho$.

Two plots P and P' are *similar* iff there is a similarity mapping ρ such that $P' = P\rho$. Likewise, two situation-goal rules $S \rightarrow G$ and $S' \rightarrow G'$ are *similar* iff there is a similarity mapping ρ such that $S' = S\rho$ and $G' = G\rho$. Given two indexed plots (S_1, G_1, P_1) and (S_2, G_2, P_2) , we say that:

- (S_1, G_1, P_1) and (S_2, G_2, P_2) are *sgp-similar* iff there is a similarity mapping ρ such that $S_2 = S_1\rho$, $G_2 = G_1\rho$ and $P_2 = P_1\rho$
- (S_1, G_1, P_1) and (S_2, G_2, P_2) are *sg-similar* iff $S_1 \rightarrow G_1$ and $S_2 \rightarrow G_2$ are similar situation-goal rules
- (S_1, G_1, P_1) and (S_2, G_2, P_2) are *p-similar* iff P_1 and P_2 are similar plots

For example, consider $I_1 = (S_1, G_1, P_1)$, $I_2 = (S_2, G_2, P_2)$ and $I_3 = (S_3, G_3, P_3)$, where

$$\begin{aligned} S_1 &= [\text{iscompof}(c3, \text{pr}), \text{ctype}(c3, \text{ct}), \text{defective}(c3)] \\ G_1 &= [\text{iscompof}(c4, \text{pr}), \text{ctype}(c4, \text{ct}), \neg\text{defective}(c4)] \\ P_1 &= [[f1: \text{order}(\text{ct}, c4), f2: \text{replace}(\text{pr}, c3, c4)], [f1-f2]] \\ S_2 &= [\text{iscompof}(c1, \text{pr}), \text{ctype}(c1, \text{ct}), \text{defective}(c1)] \\ G_2 &= [\text{iscompof}(c2, \text{pr}), \text{ctype}(c2, \text{ct}), \neg\text{defective}(c2)] \\ P_2 &= [[f7: \text{order}(\text{ct}, c2), f8: \text{replace}(\text{pr}, c1, c2)], [f7-f8]] \\ S_3 &= [\text{iscompof}(c3, \text{pr}), \text{ctype}(c3, \text{ct}), \text{defective}(c3)] \\ G_3 &= [\text{iscompof}(c3, \text{pr}), \text{ctype}(c3, \text{ct}), \neg\text{defective}(c3)] \\ P_3 &= [[f1: \text{repair}(\text{pr}, c3)], []] \end{aligned}$$

Then, the plans P_1 and P_2 are similar. Indeed, $P_2 = P_1\rho$, where ρ is the similarity mapping that maps $c4$ into $c2$ and $c3$ into $c1$. To verify whether the precedence dependencies agree, it suffices to find a renaming of the tags of one of the plots that can render the sets of precedence dependencies of the two plots equal, as discussed in Section 2.1 (note that tags were not considered when introducing similarity mapping, since they are just a notational convenience and are not required to define plots).

Likewise, the situation-goal rules $S_1 \rightarrow G_1$ and $S_2 \rightarrow G_2$ are similar, with ρ again as the similarity mapping. The indexed plots I_1 and I_2 are p-similar, sg-similar and sgp-similar. Moreover, I_3 is sg-similar to both I_1 and I_2 , but not sgp-similar.

Finally, given two plots $P=(E,D)$ and $P'=(E',D')$, we say that a plot Q is a *most specific generalization (m.s.g.)* [13,16] of P and P' iff there are similarity mappings θ and θ' such that $Q=P\theta=P'\theta'$, θ and θ' map terms into variables and θ and θ' are the least such similarity mappings. Note that Q is trivially similar to both P and P' .

For example, plot P_4 is an m.s.g. of plots P_1 and P_2 above, where

$$P_4 = [[f1: \text{order}(\text{ct}, X), f2: \text{replace}(\text{pr}, X, Y)], [f1-f2]]$$

Indeed, $P_4=P_1\theta=P_2\theta'$, where θ maps constants c_4 and c_3 into variables X and Y , respectively, and θ' maps constants c_2 and c_1 into variables X and Y , respectively (and tags f_7 and f_8 into f_1 and f_2 , respectively).

The notion of m.s.g. extends to indexed plots by applying the similarity mappings also to the circumstances. Thus, $I_4=(S_4, G_4, P_4)$ is an m.s.g. of I_1 and I_2 above, where

$$\begin{aligned} S_4 &= [\text{iscompof}(X, \text{pr}), \text{ctype}(X, \text{ct}), \text{defective}(X)] \\ G_4 &= [\text{iscompof}(Y, \text{pr}), \text{ctype}(Y, \text{ct}), \neg\text{defective}(Y)] \\ P_4 &= [[f1: \text{order}(\text{ct}, Y), f2: \text{replace}(\text{pr}, X, Y)], [f1-f2]] \end{aligned}$$

3.2 Indexed plot libraries

The data administrator (DA) may apply the plot extraction algorithm of Section 2.3 to search the database log for a plot P responding to a goal-inference rule $S \rightarrow G$. The entire set of (S_p, G_p, P) indexed plots collected by the DA constitutes a *library of plots (LP)*. Note that the LP will contain only ground indexed plots, extracted by the algorithm of Section 2.3. Furthermore, note that the LP may in fact contain similar plots, such as $I_1=(S_1, G_1, P_1)$, $I_2=(S_2, G_2, P_2)$ in the example of Section 3.1.

The DA may reduce such redundancy by replacing indexed plots which are sgp-similar by their most specific generalization, thereby constructing a *library of typical plots (LTP)*. The DA may indeed directly construct the LTP as follows. As for the LP , suppose that the DA applies the plot extraction algorithm to search the log for an indexed plot (S_p, G_p, P) responding to a previously specified goal-inference rule $S \rightarrow G$. Before adding (S_p, G_p, P) to the LTP , the DA searches the LTP for an indexed plot (S_q, G_q, Q) which is sgp-similar to (S_p, G_p, P) . If one such indexed plot is found, the DA replaces (S_q, G_q, Q) in the LTP by the most specific generalization of (S_p, G_p, P) and (S_q, G_q, Q) .

If applied to the indexed plots $I_1=(S_1, G_1, P_1)$, $I_2=(S_2, G_2, P_2)$ and $I_3=(S_3, G_3, P_3)$, listed in Section 3.1, this strategy will keep $I_3=(S_3, G_3, P_3)$ and, since I_1 and I_2 are similar, will replace I_1 and I_2 by their m.s.g. $I_4=(S_4, G_4, P_4)$, also defined in Section 3.1.

To search the LTP , an interested agent supplies two lists, L_s and L_g . The library will return any indexed plot (S_p, G_p, P) such that (L_s, L_g) *matches* (S_p, G_p) in the sense that every literal in L_s unifies with some literal in S_p , and every literal in L_g unifies with some literal in G_p . As a consequence of unification, variables (not all, in some cases) in S_p and G_p – and consequently in P – will be consistently replaced by constants present in L_s or L_g . In other words, (L_s, L_g) is a more *concrete circumstance* which must fit in the more general (S_p, G_p) circumstance to justify the use of the associated plot P , which could then be used as an executable *plan*.

For example, assume that the *LTP* contains the indexed plots $I_3=(S_3,G_3,P_3)$ and $I_4=(S_4,G_4,P_4)$, defined above. Consider the pair of lists (L_s,L_g) , where

$$\begin{aligned} L_s &= [\text{iscompof}(c3, pr), \text{ctype}(c3, ct)] \\ L_g &= [\text{iscompof}(Z, pr), \neg\text{defective}(Z)] \end{aligned}$$

Searching the *LTP* with (L_s,L_g) yields $I_3=(S_3,G_3,P_3)$ and $I_4=(S_4,G_4,P_4)$.

However, note that P_3 and P_4 , even though they were designed to operate the required transition to a state where G holds, are not equivalent with respect to their full effects. A wise precaution is to ascertain beforehand all that may be caused by running each plot, in view of possible undesired side-effects. This can be accomplished by simulating the execution of each plot, after supplying a *context*, defined as a set of literals, which captures aspects of the current state s_0 . Simulation can employ a well-known recursive backward-chaining algorithm [13], which incidentally is the basis for simple plan-generators following STRIPS formalisms.

For example, consider the context C , where:

$$C = [\text{iscompof}(c3, pr), \text{ctype}(c3, ct), \text{defective}(c3)]$$

Then, the result of choosing to apply either P_3 or P_4 would be R_3 or R_4 , where:

$$\begin{aligned} R_3 &= [\neg\text{defective}(c3), \text{ctype}(c3, ct), \text{iscompof}(c3, pr)] \\ R_4 &= [\neg\text{iscompof}(c3, pr), \text{ctype}(c3, ct), \text{ctype}(Y, ct), \\ &\quad \neg\text{defective}(Y), \text{iscompof}(Y, pr)] \end{aligned}$$

The fact that P_4 contains a variable Y , even after the simulated execution, and the fact that the variable figures in the result obtained certainly deserve attention. We can understand that, when placing the order, the foreman in charge of the product can only indicate the component type (the value of attribute ct). The value of the discriminating attribute cno will remain undetermined until the order is fulfilled, typically through the intermediacy of an agent involved with inventory management. Such problems can only be handled in multi-agent environments, a topic outside the scope of the present paper, wherein a plot would result from the combination of partial plots, each one including appropriate communicative acts [17], to be executed by different agents.

4. Plot analogy

4.1 The notion of plot analogy

The database schema introduced in Section 2.1 provides an example of weak entity. It has several typical features which recur in many other application domains. In [7], we argued that database conceptual design, especially if complex notions such as weak entities are involved, can be significantly facilitated by deriving new schemas from previously specified *analogous* schemas.

In this section, with the help of an example, we describe how to extend the analogy mapping introduced in [7] to cover domain-oriented operations, specified using pre- and post-conditions.

Given the schema of section 2.1, we first define a Weak Entity *schema pattern*, which a database designer may at any future time use to create new schemas:

```

Pattern: Weak Entity
Example scheme: Product
Clauses --

    entity(A, B)
    attribute(A, B)
    entity(C, [B/D-E-B, D])
    attribute(C, D)
    attribute(C, F)
    attribute(C, G)
    relationship(E,
        C/n/total, A/l/total)

    operation(H, [F, D])
    pre(H, [I, J], [])
    post(H, [I, J],
        [[F, J, I], [-G, J]])
    operation(K, [B, D, D])
    pre(K, [L, M, N],
        [[E, M, L], [F, M, I],
        [F, N, I]])/diff(M, N)
    post(K, [L, M, N],
        [[-E, M, L], [E, N, L]])
        /diff(M, N)
    operation(O, [B, D])
    pre(O, [L, J], [[G, J]])
    post(O, [L, J], [[-G, J]])

Mappings --
A:product
B:pno
C:component
D:cno
F:ctype
G:defective
E:iscompof
H:order
K:replace
O:repair

```

Suppose that a database designer wants to build a Team schema, involving teams and their members, and recognizes (or is told) that the intuitive mental image of the prospective schema "looks very much like" what occurs in the Product schema: members of teams are like components of products, reflecting the well-known "an organization is a machine" metaphor [22]. This motivates the introduction of the *is-like analogy mapping*, a meta-level relationship. In this example, the declaration "Team is-like Product" establishes that Product can be taken as a *source schema* on which the definition of the *target schema* Team can be partly accomplished [14].

We developed an interactive prototype tool to experiment with these notions. It prompts the designer to answer questions of the form: "What corresponds to <name>?", where <name> figures in the supposedly known source schema. The tool will use the names that the designer types to instantiate the variables in the mapping component of the Weak Entity pattern.

In our example, the mapping correspondences, to be saved for future use, are:

```

product → team
pno → tno
component → member
cno → mno
ctype → spec
defective → unprepared
iscompof → ismembof
order → hire
replace → reassign
repair → train

```

Using these mappings, the tool will then create the Team schema:

Schema: Team

```

Clauses --
    entity(team, tno)
    attribute(team, tno)
    relationship(ismembof,
        member/n/total,
        team/l/total)
    entity(member,
        [tno/mno-ismembof-tno,
        mno])
    attribute(member, mno)
    attribute(member, spec)
    attribute(member,
        unprepared)

```

```

operation(hire, [spec, mno])           /diff(B, C)
pre(hire(A, B), [])                   post(reassign(A, B, C),
post(hire(A, B),                       [-ismembof(B, A),
[spec(B,A), -unprepared(B)])          ismembof(C,A)])/diff(B,C)
operation(reassign,                    operation(train, [tno, mno])
[tno, mno, mno])                      pre(train(A, B) [unprepared(B)])
pre(reassign(A, B, C),                 post(train(A, B),
[ismembof(B, A),                       [-unprepared(B)])
spec(B, D), spec(C, D)])

```

In general, analogy mappings are not total in either direction. For some elements of the source schema, the designer may reply that there is no corresponding element in the target schema. On the other hand, after closing the dialogue, the designer may declare additional elements that are specific to the target schema. Indeed, in more semantically rich cases, it is often convenient to proceed along successive dialogues, employing a series of source schemas, which can be interpreted as an attempt to cover different *aspects* by resorting to different metaphors [18].

Once it has been so derived, the Team schema (alone or as part of the design of a larger schema) is ready to be used, employing the terminology appropriate to its distinct application domain. One can imagine that project leaders will be among the agents, instead of the foremen of the source domain.

Analogy brings in the possibility to perform comparisons, queries, etc., that go across the two domains. For brevity, we shall consider just one example.

Imagine that a certain John, with a specialty designated as `spec_s3`, and who is currently a member of team `Ta`, is found to be unprepared. What can be done in this situation? This recalls the case of component `c3` of product `pr`, when `c3` was marked as defective. Can we transpose to John what was done to `c3`?

There are two possibilities for `c3`:

$$P_1 = [[f1:repair(pr, c3)], []]$$

$$P_2 = [[f1:order(ct, c4), f2:replace(pr, c3, c4)], [f1-f2]]$$

from which the analogous plots below can be readily derived:

$$P_1' = [f1:train(Ta, John)], []]$$

$$P_2' = [f1:hire(spec_s3, Peter),$$

$$f2:reassign(Ta, John, Peter)], [f1-f2]]$$

In words, one can either submit the unprepared John for training, or look for someone else with the same specialization and perform a substitution. This kind of argument has a flavor of *case-based reasoning* [19,23,30], since it involves the adaptation of a previously used strategy to handle a different, but analogous problem.

4.2 Reusing plots from the LTP across domains by analogy

We shall now see how analogy can play a helpful role in a multi-domain environment. If one is dealing simultaneously with more than one application domain, it is necessary to insert the name of the domain in question in the *LTP* entries.

For example, suppose the *LTP* only contains entries related to the Product schema, from which the new Team schema was derived. Recall that the mapping information indicating the correspondence between names in the two schemas, gathered in the course of the derivation process, is stored for future use.

Suppose that a team leader, an agent in the mini-world of the Team schema, tries to access the *LTP* using the pair of lists (L_s, L_g) as concrete circumstance, where

$$L_s = [\text{ismembof}(\text{John}, \text{Ta}), \text{spec}(\text{John}, \text{spec_s3}), \text{unprepared}(\text{John})]$$

$$L_g = [\text{ismembof}(\text{John}, \text{Ta}), \neg\text{unprepared}(\text{John}), \text{spec}(\text{John}, \text{spec_s3})]$$

A first direct attempt to perform a match inevitably fails, since there are still no entries for Team in the *LTP*. But since it has been declared that “Team is-like Product”, and because the analogy mappings were kept, a search for analogous Product entries is automatically processed, producing the following two plots:

$$P_1 = [[f1:\text{train}(\text{Ta}, \text{John})], []] ;$$

$$P_2 = [[f1:\text{hire}(\text{spec_s3}, \text{X}), f2:\text{reassign}(\text{ta}, \text{John}, \text{X})], [f1-f2]]$$

5. Concluding remarks

We argued that plots, indexed by the situation-goal circumstances that motivate their enactment, provide a compact representation for the real-life stories happening in the mini-world of management information systems. Using the concept of similarity, we first illustrated how to organize plots from the same domain in a library of typical plots, which are ready for reuse. Then, using the concept of analogy, we discussed how to reuse such plots across different domains.

Experiments with prototype implementations, based on logic programming, have been of much help to test our approach. We are developing more robust implementations that combine the techniques outlined in this paper with an emergency response information system that will establish a transition from such early prototype tools to practical applications. More research is also required for, among other objectives, defining more complex similarity criteria, and for developing efficient methods to handle multi-agent environments.

References

1. Afanas'ev, A. *Russian Fairy Tales*. N. Guterman (trans.), New York: Pantheon Books, 1945.
2. Bae, J.; Caverlee, J.; Liu, L.; Rouse, W.B.; Yan, H. “Process Mining by Measuring Process Block Similarity”. In: Proc. Workshop on Business Process Intelligence (BPI at BPM), Vienna, 2006.
3. Bal, M. *Narratology - Introduction to the Theory of Narrative*. U. Toronto Press, 2002.
4. Barbosa, S.D.J.; Breitman, K.K.; Furtado, A.L.; Casanova, M.A. "Similarity and Analogy over Application Domains". In: Proc. XXII SBBD. João Pessoa, Brazil, Oct. 2007.
5. Barbosa, S.D.J.; de Souza, C.S. “Extending software through metaphors and metonymies”. *Knowledge-Based Systems*, 14, 2001.
6. Batini, C.; Ceri, S.; Navathe, S. *Conceptual Design – an Entity-Relationship Approach*. Benjamin Cummings, 1992.
7. Breitman, K.K.; Barbosa, S.D.J.; Casanova, M.A.; Furtado, A.L. “Conceptual modeling by analogy and metaphor”. In: Proc. CIKM 2007. Lisbon, Portugal. Nov. 2007.

14 **Antonio L. Furtado, Marco A. Casanova,
Simone D.J. Barbosa, Karin K. Breitman**

8. Carvalho, M.T.; Freire, J.; Casanova, M.A. "The Architecture of an Emergency Plan Deployment System". In: Proc. III Workshop Brasileiro de GeoInformática, Rio de Janeiro, Brasil, pp. 19-26, Oct. 2001.
9. Ciarlini, A.E.M.; Furtado, A.L. "Understanding and Simulating Narratives in the Context of Information Systems". In: Proc. 21st. International Conference on Conceptual Modeling, Tampere, Finland, Oct. 2002.
10. Ciarlini, A.E.M.; Veloso, P.A.S.; Furtado, A.L. "A Formal Framework for Modelling at the Behavioral Level". In: Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, 2000.
11. Fernandes, A.; Ciarlini, A. E. M.; Furtado, A. L.; Hinchey, M. G.; Breitman, K. K.; Casanova, M. A. "Adding Flexibility to Workflows through Incremental Planning". *Innovations in Systems and Software Engineering*, 2007.
12. Fikes, R.E.; Nilsson, N.J. "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence*, 2(3-4), 1971.
13. Furtado, A.L.; Ciarlini, A.E.M. "Constructing Libraries of Typical Plans". In: Proc. 13th Int. Conf. on Computer Advanced Information System Engineering, 2001.
14. Holyoak, K.; Thagard, P. *Mental Leaps*. The MIT Press, 1996.
15. Kautz, H.A. (1991) "A Formal Theory of Plan Recognition and its Implementation". In: Allen, J. F. et al (eds.) *Reasoning about Plans*. Morgan Kaufmann, 1991.
16. Knight, K. "Unification: A Multidisciplinary Survey". *ACM Comp. Surveys*, 21(1), March, 1989.
17. Labrou, Y.; Finin, T. "History, State of the Art and Challenges for Agent Communication Languages". In *Informatik – Informatique* 1, 2000.
18. Lakoff, G.; Johnson, M. *Metaphors We Live By*. U. Chicago Press, 1980.
19. Leake, D. *Case-Based Reasoning*. The MIT Press, 1996.
20. Lippert, M.; Schmolitzky, A.; Züllighoven, H. "Metaphor Design Spaces". In: Proc. 4th Int. Conf. Extreme Programming and Agile Processes in Software Engineering, XP 2003. Genova, Italy, May 25-29, 2003.
21. Lo, D.; Khoo, S-C.; Liu, C. "Efficient Mining of Recurrent Rules from a Sequence Database". Proc. 13rd Int. Conf. on Database Systems for Advance Applications (DASFAA'08). New Delhi, India. March 19-21, 2008 (to appear).
22. Morgan, G. *Images of organization*. Sage Publications, 1998.
23. Muñoz-Avila, H.; Cox, M. "Case-based plan adaptation: An analysis and review". *IEEE Intelligent Systems*, 2007.
24. Propp, V. *Morphology of the Folktale*. Laurence, S. (trans.), Austin: U. Texas Press, 1968.
25. Schank, R. *Tell me a Story*. Northwestern University Press, 1990.
26. Turner, M. *The Literary Mind*. Oxford University Press, 1996.
27. Van de Walle, B.; Turoff, M. "Emergency response information systems: emerging trends and technologies special section – Introduction". *Comm. of the ACM*, 50(3), pp.29-31, March 2007.
28. van der Aalst, W.M.P.; Weijters, A.J.M.M.; Maruster, L. "Workflow Mining: Discovering Process Models from Event Logs". *IEEE Transactions on Knowledge and Data Engineering*, 47(2), pp. 237-267, 2004.
29. Winston, P.H. "Learning and reasoning by analogy". *Comm. of the ACM*, 23, pp. 689-703.
30. Yang, Q.; Cheng, H. "Case Mining from Large Data Bases". Proc. 2003 Int. Conf. on Case-based Reasoning. Trondheim, Norway, 2003.