Extending software through metaphors and metonymies

S.D.J. Barbosa*, C.S. de Souza

Computer Science Department, Pontifícia Universidade Católica, R. Marquês de São Vicente, 225 — Gávea, Rio de Janeiro, RJ 22453-900, Brazil

Abstract

This article is about applications that can be customized or extended through their own user interface. This is achieved by the interface’s ability to interpret users’ non-literal expressions, namely metaphorical and metonymic ones. Such increased interpretive intelligence depends on static and dynamic models of the domain and application, from which new figurative meanings are abducted automatically or semi-automatically. The system performs controlled modifications on the underlying models, based on inferences it draws about users’ intentions as they produce figurative utterances. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: End-user programming; Metaphor; Metonymy; Abductive reasoning; Interfaces for knowledge-based systems; Semiotic engineering

1. Introduction

Academia and industry alike have recognized the need to create extensible software in order to allow users to tailor applications to their particular needs and alleviate the programming effort involved in software upgrades [3,12,13,25,26]. However, commercially available extensible applications have typically suffered from rigid and unhelpful interfaces which present a number of obstacles to achieving this goal. One of the major difficulties for extending software is the user’s lack of knowledge about the underlying application model and about how to turn familiar interactive patterns into function-triggering protocols.

The most popular extension mechanisms to date have focused primarily on the automation of repetitive tasks (through different techniques, such as macro recording and programming by demonstration [5]). However, the role of extensions is not limited to that of circumventing repetitions. It is also related to creative usage of software in innovative contexts, for which scripting and programming languages have been required typically.

In this paper, we present a mechanism for extending software that can cope with the typical incomplete and imprecise knowledge users have about the applications they interact with. It can automatically or semi-automatically generate extensions to accommodate both repetitive and innovative tasks within a controlled range of the application’s domains. By assigning meaning to metaphorical and metonymic expressions, our extension mechanisms recognize meaningful constructs beyond those syntactically acceptable in terms of the original models produced by the application’s designer. In this way we achieve a kind of programming while interfacing.

As figures of speech, metaphors and metonymies are best explored in language-based interfaces, where users have a wider range of linguistic resources to express themselves. However, although to a lesser extent (considering that visual or gestural metaphors and metonymies are themselves extended concepts we must be prepared to cope with), graphic user interfaces can benefit from our approach, as we will briefly illustrate in the following sections.

We are committed to a semiotic engineering perspective [8,10], according to which an application is a one-shot message sent from system designers to system users, about how to send and receive other messages in order to achieve a certain range of goals. The major challenge for extensible applications in this perspective is to find out how the designers should tell users what can be extended, and how this can be done.

Typical users have a fairly incomplete or even incorrect knowledge about underlying application models and the way in which such models are put to work through human–computer interaction. For this reason, treating users as programmers (i.e. as people who can produce a working semantic model of an application) has been the origin of much frustration and untapped functionality.
In our approach, we will assume that the underlying semantic models of the application are equivalent to the semantic models implied or presupposed in the spectrum of conversations that have been rationally designed to take place between user and system. In other words, whatever the users infer about the application by way of the interaction they are able to have with the system legitimately belongs to a semantic model equivalent to that of the original designer’s.

Following the semiotic engineering perspective, we will tell users that the way to extend the application as they perceive it is analogous to a familiar way we all use to extend somebody’s vocabulary and/or grammar: by comparison with terms that are previously known by both parties. Thus, extending applications is tantamount to learning new meanings in everyday life.

It is essentially important that designers communicate effectively their interpretations and assumptions about the domain of the application and its potential contexts of use in order to increase the quality of the match between the users’ models and their own. In an extensible application, this is critical because users will assume the role of designers, however limited. It is also essential that designers tell users that certain metaphors and metonymies can be interpreted by the system, which can cause extensions to the original application model. Therefore, such an application must be conceived for communicating knowledge relevant to the design task as well.

Our approach is not an end-user programming (EUP) mechanism in the most popular sense, but rather a particular kind of linguistic mechanism that achieves the same effect as practical extensions. We call it extension by interpretation [1]. The interface language allows users to produce metaphorical or metonymic expressions and the language interpreter attempts to make sense of such expressions. When it succeeds, it behaves according to the interpretation it derived; when it doesn’t, it interacts with the user based on partial interpretations. If no interpretation is derived, the interface language produces no sense of what the user is saying and reacts as any typical interface in non-extensible software.

The sense-making process is a kind of abductive process [29,18] that generates possible interpretations to novel expressions by means of specific metaphorical and metonymic operators. Compared to alternative approaches that try to meet major EUP cognitive challenges by progressively disclosing commands and programming structures [11,12], ours allows users to achieve changes without even knowing that what they want to do is innovative in any sense. This feature is also distinctive in comparison to most programming by demonstration approaches [5].

Nevertheless, not knowing about extensions may have undesirable side effects, especially in view of figurative speech. This kind of expression is generally used in association with rhetorical effects that are only pertinent to very specific contexts of use. Consequently, we make a distinction between vanishing and permanent extensions. Vanishing ones result from interpreting figurative utterances and performing the inferred actions on discardable extended copies of the original models. Permanent ones require that the users be made aware of extensions and carefully decide about the need or convenience of replacing the original models with the extended ones, in a similar process to that of extending a natural language’s lexicon with dead metaphors and metonymies.

This paper concentrates on vanishing extensions. Our proposed model has been partially implemented and tested for conceptual consistency in three separate modules: the intelligent UI knowledge base (KB) and reasoner, the graphic environment, and a preliminary knowledge acquisition interface [1,4]. In the following, we will disclose and discuss aspects of our system’s models and reasoning processes, and provide examples of interpreting vanishing non-literal expressions based on a groupware application designed to support the organization and usage of collective knowledge. In the last section of the paper we discuss the innovative aspects of our approach in view of the representation and modeling costs that it may entail.

2. Extensions through analogies

Many researchers in the field of cognitive science agree that we humans think and express ourselves extensively in non-literal ways [21,23,27]. In particular, we make use of metaphors and metonymies in order to understand or explain a concept in terms of others, by highlighting a concept’s characteristics or relations, and concealing others. It is also known that, in order to effectively use this kind of figurative language in our communication, it is necessary for sender and receiver to share knowledge, assumptions, and cultural background [23].

In computer applications, a fundamental step to dealing with conceptual metaphors is that the designer’s knowledge and assumptions be communicated to users in a consistent interface language [2,7]. For sake of computational efficiency, a balance must be found between interface language and application language, so that both are good enough for man and machine.

It is impossible to represent, computationally or not, all common sense knowledge that arises from our experience in the world. Thus, computer applications should not be expected to behave like partners in a natural communicative process, capable of negotiating meanings until the parties reach mutual understanding. Rather, computer users should be aware that they are interacting with a symbolic artifact created by a human designer, who consciously embedded in it only part of his or her knowledge and assumptions about the application domain and the users’ profile [9].

One of the hardest EUP challenges is related to the users’ lack of knowledge about such embedded application and domain models. By drawing on the works of French [14] and Holyoak and Thagard [20], our approach helps fill this
knowledge gap with an enhanced representation of domain and application models, which are manipulated by abductive mechanisms that interpret metaphorical and metonymic utterances. These utterances often account for what can be diagnosed as imprecise and incomplete knowledge in traditional approaches.

When a user is familiar with a portion of the application and has built a partial conceptual model of it, he or she is likely to try to make sense of unfamiliar portions in terms of known concepts. So, in order to help users acquire a better understanding of the application, we calculate possible interpretations for their utterances and give them feedback about the reasoning process involved in each interpretation. This unfolds to the user aspects of the internal application structure.

Although we might automatically extend the application when an interpretation is found, underlying model adjustments can certainly have undesirable side effects. Our model distinguishes between vanishing and persistent extensions. We keep the status of metaphorical or metonymic expressions as figurative speech until they are explicitly integrated to the domain discourse with their corresponding syntactic and semantic specifications.

One way to achieve explicit integration is to prompt the user after several sessions of interaction with the application, when identical patterns of metaphorical and metonymic expressions arise. The application may offer users the option to make them persistent, as a computational equivalent of turning live metaphors into dead metaphors. In the next section we will describe representations and computations involved in interpreting metaphor and metonymy.

3. Modeling for metaphors and metonymies

Calculating metaphors and metonymies requires special representation structures — static and dynamic domain and application models, enriched by metaphorical and metonymic chain classifications. In order to describe how the domain and application models must be represented, we first need to understand what a user utterance is, and what variations of literal utterances we want to interpret.

A user utterance is a syntagmatic expression [30], a linearized rule-based combination of elements in the user interface language. The interaction paradigm determines the form of the expression, like object + verb and verb + object, for instance. A literal utterance has a clear, direct and unambiguous interpretation, derived from the language grammar.

A metonymic utterance occurs when reference to an element is made by another element with which the first has a relation of part–whole, content–container, cause–effect, producer–product, among other possibilities. For example, when we say “He’s got a Picasso”, we mean he’s got a work of art produced by Picasso. In a computer application, we might express (possibly by means of push-buttons or mouse pointer movements of a typical GUI interface) “copy the boldface”, to mean “copy the text formatted in boldface”. Still regarding computer applications, metonymies can also be used to generate iterations and recursions. For instance, in a graphical editor that allows users to group objects, if a user selects a group and chooses a different fill color, it iterates through all elements in the selected group and applies the chosen fill color to each element that can be filled, individually. This is clearly figurative speech, where the user is saying to color a compound element, meaning of course that only the “colorable elements” should be affected. Moreover, if an element is itself a group, the operation is performed on its elements, too, recursively. Nevertheless, such usage of metonymies is typically ad hoc, or incidental, not to be consistently found elsewhere in the application. Users must learn where such metonymies may be used in isolation, for they cannot predict the behavior of seemingly analogous situations.

In order to be able to generate metonymic expressions, designers need to represent relations among elements and identify which ones may be part of a metonymic chain. Composition and aggregation relations, such as part-of, are natural candidates for metonymy. Other relations must be explicitly declared as having metonymic potential, such as relations representing location, ownership, possession, creation, and many others.

For each element in a non-literal expression, we traverse the metonymic chains in the static domain model, making a paradigmatic substitution and checking if the resulting expression has a literal interpretation. A valid substitute becomes the metonymic target. The utterance interpretation is the result of an iteration through every element in the original expression obtained by following the chain to the metonymic target, or every metonymic target reached from the original expression, depending on the direction traversed.

There are two possible directions of traversal: (1) from “whole” or “producer” to “part” or “product”, and (2) from “part” or “product” to “whole” or “producer”. For instance, in a bibliographical domain, if we ask to copy “Machado de Assis”, all references corresponding to literary works of “Machado de Assis” would be copied (whole-to-part direction of traversal). If, on the other hand, we have the relation “Capitu” is a character in “Dom Casmurro”, and ask to copy “Capitu”, the reference to the book “Dom Casmurro” would be copied (part-to-whole direction of traversal).

The preferential direction of traversal (whole-to-part or part-to-whole) is a parameter for the algorithms, which can be tuned to specific language profiles. This is so because figurative speech may not (and usually does not) translate across languages. All of the examples illustrated here are fine-tuned to Brazilian Portuguese.

Metaphors may arise when comparing the relations between pairs of elements. For example, there may be a relation “written by” linking a text to its author, and the instances “O Cortiço written by Aluísio de Azevedo”, and
“O Guarani written by José de Alencar”. The expression “Aluísio de Azevedo’s O Guarani” will result in retrieving an instance of “Something written by Aluísio de Azevedo”. If the underlying domain representation is rich enough to single out Alencar’s O Guarani as his most famous novel, the metaphorical representation can qualify “something written by” with the attribute “most famous”, and thus retrieves Azevedo’s novel that is, in the representation, as remarkable as O Guarani is for Alencar. We thus see that this interpretation is at the boundaries of poetic use. The appropriateness and sophistication of interpretations is directly proportional to the expressiveness of the underlying domain models.

Our mechanism for generating metaphors can also be used to create synonyms that express the designer’s idea of equivalence in particular contexts. For instance, in an academic institution there might be a “text” classification including “paper”, “article”, and “report”. A systematic interpretation of these words in terms of each other is equivalent to neutralizing the distinctions among them, and making them interchangeable in some or in all contexts, as singled out by Lakoff in his theory [22].

Many researchers have investigated the computation of analogies [14–16,19,20]. Our work is based on Holyoak and Thagard’s criteria of similarity and structure [20]. When a non-literal utterance is encountered, the application will first look for classifications in which the elements occurring in the utterance may be substituted by a similar token, and check if the resulting utterance is valid. If the utterance involves two types A and B, we look for structural similarity that matches the classical analogy model A:X < Y:B. If many different alternatives are found, the mechanism looks for similarities among the attributes of the eligible types. The most similar candidate to the type occurring in the original utterance will then be selected for the replacement.

We may use classifications, relations, and attributes to generate and disambiguate metaphorical interpretations for users’ non-literal expressions. However, when it is impossible to disambiguate terms or when there are many alternatives for interpretation, the application should present choices to users, along with an explanation about how they were generated, and have users select the one they mean, or discard them all and try to use another form of expression.

3.1. Interpretation mechanisms for metonymies

The following PROLOG predicates illustrate our mechanism for calculating metonymies:

```prolog
% resolve_reference (Instance, ExpectedType, List, Description):=
% consider chains of length 1 or 2
resolve_reference(Instance, ExpectedType, List, Description):-
instance_of(Instance, _, _),
entity(ExpectedType, _),
get_metonymy(Instance, ExpectedType, List, 1, Description).
resolve_reference(Instance, ExpectedType, List, Description):-
instance_of(Instance, _, _),
entity(ExpectedType, _),
get_metonymy(Instance, ExpectedType, List, 2, Description).
% consider both normal and inverse directions
get_metonymy(Instance, ExpectedType, List, Length, Description):-
metonymic_chain(_, Rel, Instance, Length),
findall(NewInstance, (metonymic_chain(NewInstance, Rel, Instance, Length), subinstance(NewInstance, ExpectedType)), List),
not(length(List,0)),
swriteln(Description, "metonymy size %w: %w", [Length, Rel]).
get_metonymy(Instance, ExpectedType, List, Length, Description):-
metonymic_chain_inverse(_, Rel, Instance, Length),
findall(NewInstance, (metonymic_chain_inverse(NewInstance, Rel, Instance, Length), subinstance(NewInstance, ExpectedType)), List),
not(length(List,0)),
swriteln(Description, "metonymy inverse size %w: %w", [Length, Rel]).
```

The following PROLOG predicates illustrate our mechanism for calculating metonymies:

```prolog
% resolve_reference (Instance, ExpectedType, List, Description):=
% metonymical interpretation, by instances
```
NewN is N + 1.

metonymic_chain_inverse(To, [From, Rel, To], From, 1) :-
    instance_of(From, Rel, To, _),
    metonymic_relation(Rel, _).

metonymic_chain_inverse(To, [From, Rel1 | Rel], From, NewN) :-
    instance_of(From, Rel1, X, _),
    metonymic_relation(Rel1, _),
    metonymic_chain_inverse(To, Rel, X, N),
    NewN is N + 1.

3.2. Interpretation mechanisms for metaphors

The following Prolog predicates illustrate our mechanism for interpreting metaphors:

% ±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±
% on attributes:
% resolve_reference(value, (Value), (ISource), (expected type), (list of matched instance(s)), (description))

resolve_reference(value, Value, ISource, Output, Description) :-
    instance_of(ISource, ExpectedType),
    value(Value, Attrib, _, _),
    findall([ITarget, Common], (subinstance(ITarget, ExpectedType), value(Value, Attrib, ITarget, _),
        not(value(Value, Attrib, ISource, _)),
        common_attributes(ITarget, ISource, Common), List),
    not(length(List, 0)),
    findall(T, member([T, _], List), Output),
    findall(U, member([_, U], List), OutputCommon),
    swriteln(Description, "metaphor 4 (attributes): a(n) %w %w -> %w.%w=%w", [Value, ISource, Output, Attrib, Value]).

resolve_reference(value, Value, ISource, Output, Description) :-
    instance_of(ISource, ExpectedType),
    value(Value, Attrib, _, _),
    findall(ITarget, (subinstance(ITarget, ExpectedType),
        value(Value, Attrib, ITarget, _),
        not(value(Value, Attrib, ISource, _)), Output),
    not(length(Output, 0)),
    swriteln(Description, "metaphor 3 (attributes): a(n) %w %w -> %w.%w=%w", [Value, ISource, Output, Attrib, Value]).

% ±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±
% on relations:
% resolve_reference((Ref), of, (Source), (expected type), (list of matched instance(s)), (description))
% or
% resolve_reference((Source), s, (Ref), (expected type), (list of matched instance(s)), (description))

resolve_reference(Ref, s, Source, Output, Description) :-
    resolve_metaphor(ISource, IRRef, Output, Description).

resolve_metaphor(IA, ID, IC, Description) :-
    A:B :: C:D
    instance_of(IA, X, _),
    instance_of(ID, Y, _),
    X \= Y, %source and ref should be of different types
    subinstance(IA, TA),
    subinstance(ID, TD),
    subinstance(IC, TA), IC \= IA,
    subinstance(IB, TD), IB \= ID,
    metaphor(IA, IB, IC, ID, Description).

metaphor(A, B, C, D, Description) :-
    relation_chain(A, Rel, B, Attrib),
    relation_chain(C, Rel, D, Attrib),
    not(length(Attrib, 0)),
    describe_relations(Rel, RelDesc),
    swriteln(Description, "metaphor 1ab-cd (relations and relation attribs): (%w %w %w %w) and (%w %w %w %w) *** %w", [A, Rel, B, Attrib, C, Rel, D, Attrib, RelDesc]).

metaphor(A, B, C, D, Description) :-
relation_chain(B, Rel, A, Attrib),
relation_chain(D, Rel, C, Attrib),
not(length(Attrib, 0)),
describe_relations(Rel, RelDesc),
swritef(Description, "metaphor 1 ba–dc
(relations and relation attribs): (%w
%w %w %w) and (%w %w %w %w) *** %w", [A,
Rel, B, Attrib, C, Rel, D, Attrib,
RelDesc]).

metaphor(A, B, C, D, Description):-
relation_chain(A, Rel, B, Attrib1),
relation_chain(C, Rel, D, Attrib2),
(length(Attrib1, 0); Attrib1 = Attrib2),
describe_relations(Rel, RelDesc),
swritef(Description, "metaphor 2 ab–cd
(relations): (%w %w %w %w) and (%w %w %w
%w) *** %w", [A, Rel, B, Attrib1, C, Rel,
D, Attrib2, RelDesc]).

%------------------------------------------
% AUXILIARY TRAVERSAL FUNCTIONS
%------------------------------------------
% on entities
%------------------------------------------
relation_chain(From, [Rel], To):-
relation(From, Rel, To, _).
relation_chain(From, [Rel|T], To):-
relation(From, Rel, X, _),
relation_chain(X, T, To).

% on instances
% relations without attributes
%------------------------------------------
relation_chain(IFrom, [Rel|T], ITo, []):-
instance_of(IFrom, Rel, IFrom, _).
relation_chain(IFrom, [Rel|T], ITo, [Attrib]):-
instance_of(IFrom, Rel, X, _),
relation_chain(X, T, ITo, Attrib),
not(member(Rel, T)),
not(member(inv(Rel), T)).

% relations with attributes
%------------------------------------------
relation_chain(IFrom, [Rel], ITo, [Attrib|Value]):-
instance_of(IFrom, Rel, X, AttribList, _),
member([Attrib, Value], AttribList).
relation_chain(IFrom, [Rel], ITo, [[Att|Value]|TA]):-
instance_of(IFrom, Rel, X, AttribList, _),
member([Attrib, Value], AttribList),
relation_chain(X, T, ITo, TA).

describe_relations(Rel, mixed):-
member(X, Rel),
not(metonymic_relation(X, _)).
describe_relations(_, metonymic):- !.

subinstance(I, Type):-
instance_of(I, Type, _).
subinstance(I, Type):-
instance_of(I, TypeInt, _),
subtype(TypeInt, Type).

common_attributes(I1, I2, List, Minimum):-
findall([V1, A1], value(V1, A1, I1, _), List1),
findall([V2, A2], value(V2, A2, I2, _), List2),
intersection(List1, List2, List),
length(List, Length), Length > = Minimum.

We acknowledge that there are very sophisticated
mechanisms for calculating metaphors [14,19,20], including
some applied to the interpretation of literary texts [16].
However, we are dealing here with simple instructions to a computer application, uttered by a user. As such, we believe users will be as direct as they can, and therefore we are able to employ simplified mechanisms and yet obtain significant results. The focus is to use this kind of resource, together with a KB, in order to try to make sense out of users’ less precise, incomplete, or even incorrect utterances. In case we need a more sophisticated mechanism for interpreting metaphors, we may substitute it for the one presented in this paper.

The abductive mechanisms described here for generating metaphors and metonymies are generic, but they are applied to domain-specific models. They may be used in a variety of domains, but it is the richness of representation that will determine the opportunity for interesting abductions. However, they are not independent of interaction style. They are strongly dependent on the level of articulation (i.e. on how many parts constitute what sorts of structure in the language) and expressiveness (i.e. on what contents can be conveyed by linguistic forms) of the interface language(s). For instance, a highly visual direct manipulation interface typically has a low level of articulation (because icons and images cannot be decomposed into many and varied meaningful constituents that can combine into higher-order structures). A command language, however, may offer a finer level of articulation and higher expressiveness (because of natural-language-based regularities such as the morphology of number markers and the conjunction of phrasal constituents). Researchers have shown that a combination of visual and textual language styles maximizes the benefits and helps overcome such limitations [24]. The next section will illustrate how these mechanisms can be used to augment user interface language expressiveness in the prototype application we have partially implemented.

4. Understanding users: a sample case

Members of our research group are developing an environment to support the organization of collective knowledge produced and used by our small community. At this stage, the system is expected to help users classify documents and make annotations about them. One of the main features of our system is the ability to structure annotations about a set of documents into a thread of group discussion discourse, according to some pre-established rhetorical relations they may have with each other.

For the purposes of this paper, we are going to illustrate our mechanisms by means of a few brief extensions on the system’s model and KB. Fig. 1 illustrates a partial domain model of the domain.

The predicates corresponding to the sketched model are:

- is_a(document, text)
- is_a(annotation, text)
- relation(text, written_by, user)
- relation(text, annotated_by, annotation)
- relation(user, is_expert, field, [level_of_expertise])
- relation(user, belongs_to, group)
- relation(document, addresses, field)
- attribute(name, user)
- valid_values(level_of_expertise, [low, average, high])

As we mentioned earlier, static and dynamic models support the interpretive processes. The static model may present not only inheritance (is_a) and composition (part_of) relations, but any static relations the designer should choose to represent. In Fig. 1 we see that besides is-a relations represented by the triangle, we have the relations written-by, annotated-by, addresses, is-expert and belongs-to. The represented types may also have attributes, such as name, which we have chosen not to represent in the diagram for sake of clarity.

The designer must define which relations in the static model can be used within metonymic chains. Composition (part_of) relations are chosen by default. In this example, the designer chose the belongs-to, written-by and annotated-by relations, as indicated by the gray areas.

The dynamic model contains the operations upon elements in the static model. A few operations in our sample domain are illustrated as follows:

- list (text)
- view (text)
- annotate (text)

Some classifications are implicitly defined in the static model, by inheritance relations (is_a). These relations allow us to derive a few other operations, such as:

- list (document)
- view (document)
- annotate (document)
- annotate (annotation)

However, we often need additional classifications in
order to obtain more sophisticated metaphorical utterances. Fig. 2 presents two sample classifications in this domain.

Combining the static and dynamic models, on the one side, with the classifications provided by designers, on the other, we now turn to the examples that illustrate metaphorical and metonymic expressions that can be interpreted.

**Example 1** (Creating an entity). Our first example creates a new entity based on an existing one. Let us consider the following utterance: “A binder is a group of documents”. It could be rewritten as the following predicate:

\[
\text{analyze_sentence(binder, is, group, of, document)}
\]

In this case, since group is an entity and binder does not exist, the interpretation is that the user wants to create a new entity called binder, taking on attributes and relations of the group entity. There is only one alternative (Fig. 3), which is:

- create an entity binder, and a metonymic relation text-belongs-to-binder

The model is then extended as shown in Fig. 4. The new entity, binder, can now be instantiated, and used in metaphorical or metonymic utterances. For instance, if documents A–C belong to a binder X, then read(X) and annotate(X) generate an iteration through the three documents, and execute the corresponding operation for each document.

**Example 2** (Creating an entity with an additional attribute). Our next example considers the utterance: “An evaluation is an annotation with a grade”. This may be considered an analogy, and rewritten as:

\[
\text{analyze_sentence(evaluation, is, annotation, with, grade)}
\]

We cannot a priori decide whether grade should be considered an attribute of the new entity evaluation, or another entity which is part of it. Therefore, the user should be presented with the alternative options and be asked to choose one of them:

- grade is an attribute of evaluation
- grade is part of evaluation

The resulting model would include the entity evaluation as a third subtype of text, and it would also include the relation annotated-by from text to evaluation. Once again, the user should be able to change the name of this relation (e.g. to evaluated-by), or even remove it (Fig. 5).

Moreover, the operations on annotation may also be used for evaluation. However, these operations do not consider the new grade attribute or part, and must be modified to deal with this new element. The extension
mechanism should guide users through this process of deciding which operations remain as they are, and which ones should be modified and how. This process of modifying existing operations is dealt with elsewhere [6].

**Example 3** (Creating a relation attribute). Another example illustrates the creation of an association between two entities, also based on a metaphor. Let us consider the following user’s utterance: “Complexity is a document’s level of expertise”. It could be rewritten as the following predicate:

\[ \text{analyze_sentence(} \text{complexity, is, level_of_expertise, of document)} \]

In this case, level of expertise is a relation attribute. There is more than one alternative, however, and the user should be prompted to make the disambiguation (Fig. 6):

- create a relation user \( \rightarrow \) is-expert \( \rightarrow \) document, with an attribute named complexity, with valid values low, average, and high
- create a relation document \( \rightarrow \) is-expert \( \rightarrow \) user, with an attribute named complexity, with valid values low, average, and high
- associate an attribute named complexity to the existing relation document \( \rightarrow \) addresses \( \rightarrow \) field, with valid values low, average, and high

In case the user chooses to create a new relation, he should also be allowed to rename it. In our example, he chose to add the complexity attribute to the existing addresses relation. However, he cannot rename this relation, for it belongs to the original designer’s model.

Having finished the extension, the user may now start classifying documents in terms of the complexity with which they address a certain field. Then, he could refer to these documents using the new qualifications, such as “introductory documents”. This is possible because introductory and low belong to a common classification (as illustrated by Fig. 2), and low is a valid value for the complexity attribute related to document.

For the next couple of examples, we will consider the following instances and relations:

\[
\begin{align*}
\text{% instances of user} \\
\text{instance_of(} \text{\textquote{John_Kippling}, user)} \\
\text{instance_of(} \text{\textquote{Mike_Freeman}, user)} \\
\text{instance_of(} \text{\textquote{Paul_Winston}, user)} \\
\text{\textquote{Logic}, field)} \\
\text{instance_of(} \text{\textquote{Formal Languages}, field)} \\
\text{instance_of(} \text{\textquote{Software Engineering}, field)} \\
\text{\textquote{John_Kippling}, is_expert, Logic, high)} \\
\text{relation(} \text{\textquote{Mike_Freeman}, is_expert, ‘Formal Languages’, high)} \\
\text{relation(} \text{\textquote{Paul_Winston}, is_expert, ‘Formal Languages’, low)}
\end{align*}
\]

**Example 4** (Metaphorical interpretation of a query). Let us now consider a query represented by the following metaphorical utterance: “Who is the John Kippling of Formal Languages?”

\[ \text{analyze_question(who, is, ‘John_Kippling’, of, ‘Formal Languages’) } \]
Our interpretation mechanisms would find the instance:

\[ \text{relation('Mike_Freeman', expert_in, 'Formal Languages', high)} \]

However, if we had an additional predicate

\[ \text{relation('John_Kippling', expert_in, 'Software Engineering', low)} \]

The user would be left with two alternative options to choose from:

- Mike_Freeman \(\rightarrow\) expert-in \(\rightarrow\) Formal Languages with level_of_expertise \(\hat{=}\) high
- Paul_Winston \(\rightarrow\) expert-in \(\rightarrow\) Formal Languages with level_of_expertise \(\hat{=}\) low

In this case, were we to keep track of a history of the users’ utterances, we could establish a preferential interpretation, and even avoid the need for the user’s disambiguation. This example showed how a user with an incomplete understanding of the domain may be able to acquire knowledge from a few known elements.

Up to this point, we have seen examples of interpretation upon the static model only. We now turn our attention to possible figurative usage of the operations defined in the dynamic model.

**Example 5** (Iteration from metonymic utterances). An example of metonymic extension may be obtained by the utterance “annotate Logic”. At first, Logic will be matched to the corresponding type: field. However, the operation annotate expects a text operand, and not field. Fig. 7 illustrates some of the paths traversed in the model in an attempt to provide a valid interpretation to this utterance.

As a result, the user will be prompted with a few alternative operations he can choose. If the language provides mechanisms to state these alternatives in a literal way, then the corresponding literal constructs should also be presented, so that users may learn and later provide a syntactically correct utterance when necessary.

There are two ways in which users may benefit from metaphors and metonymies. One is that expert users may wield these mechanisms as a more efficient way of communication, which can serve rhetorical purposes such as focusing on some aspects of objects in detriment of others. For example, saying “read(John Kippling)” may serve to express the user’s focus on the author of documents in detriment of their subject field or complexity. The other is that they may gain more understanding about the underlying domain and application models if interpretive chains are unfolded and combined with explanations for the system reasoning. This would be used primarily by naïve users that are unfamiliar with large portions of the application. Explanations could incorporate the models, the operations, and the utterances involved in such abductive processes, and could make use of textual languages and visual representations. The designer should, however, be careful with the level of interference when designing for disclosure and explanations. A delicate balance between eagerness and obtrusiveness should be reached, in order to keep naïve
users well informed about the interpretations without hindering efficiency as a whole.

5. Extensibility

Our approach brings about some relevant changes relative to EUP techniques. Users may express themselves in a non-literal fashion, and achieve extensions without necessarily knowing they are doing so. In this section, we present examples of literal and non-literal expressions, in order to show the inferencing capabilities of our mechanisms.

The literal constructs provided for creating subtypes, aggregates and relations are:

1. \((A) \text{ is a } (B)\) (used to create a subtype of B)
2. \((A) \text{ is a collection of } (B)\) (used to create an aggregate)
3. \((A) \text{ (some-relation) } (B)\) (used to create a relation between A and B)
4. \((A) \text{ is like a } (B) \text{ with } (P)\) (inheritance in addition to some part or attribute)
5. \((A) \text{ is like a } (B) \text{ without } (Q)\) (used to create a clone of type A, and then remove some part or attribute)
6. \((A) \text{ is like a } (B) \text{ with } (P) \text{ instead of } (Q)\) (used to create a clone of type A, and then substitute some part or attribute)

Therefore, we may have literal utterances like the following:

- an evaluation is a text (inheritance)
- a binder is a collection of texts (aggregation)
- a user reads a text (relation only)

In order for an utterance to be literal, syntactic and semantic rules must be followed. If there is any variation to these, the utterance is considered to be non-literal, and the metaphorical and metonymic interpretation mechanisms are triggered.

When a word or expression is found that cannot be understood in the provided context, the extension mechanism verifies if that word would be valid in any other context, and then extracts from the valid context the appropriate elements and structure needed to make sense of it in the actual context. This consists of a mechanism to generate polysemy, or related meanings to a single word, one of the evidences of usage of metaphors presented by Lakoff [22].

Further evidence presented by Lakoff may also be accounted for. Previous abductive processes used to interpret metaphor and metonymy may be stored in order to be used as inference patterns that can be potentially generalized and used in mappings with a different domain. Moreover, our mechanisms offer ways to create types and meanings. Consider the following lines, with excerpts of interaction in a command language environment. The letter U indicates a user’s utterance, and the letter A indicates an application’s response. In it, a morphological rule of many languages (namely that plurals are marked with the suffix ‘-s’) is effective to facilitate representation.

\begin{verbatim}
U: a critique is like a group, but with texts and citations
A: what is citations (type or attribute)?
U: citation is a type
A: "citations" means one or more than one "citation"?
U: yes
A: a critique is like a group and like an annotation

In this case, the user must be prompted to select which attributes, parts, and relations are valid and which aren’t, by taking as a starting point all elements related to groups and annotations. If only one type is mentioned in the user’s utterance, the application may provide an interpretation, tell the user it has done so and proceed, allowing the user to intervene if he finds it necessary. For instance:

U: an evaluation is an annotation with a grade
A: what is grade? (type or attribute)
U: attribute
A: what are the possible values of "grade"?
U: 1,2,3,4,5
A: are 1,2,3,4,5 disjoint values?
U: yes

A more direct interface may be provided, with wizards and dialogs that present all the alternatives for a given concept at a glance, with one of them selected by the application as a default. This interactive style is more useful when we have multiple types referenced in an utterance, such as the following example:

U: a critique is like a group and like an annotation

In this case, the user must be prompted to select which attributes, parts, and relations are valid and which aren’t, by taking as a starting point all elements related to groups and annotations. If only one type is mentioned in the user’s utterance, the application may provide an interpretation, tell the user it has done so and proceed, allowing the user to intervene if he finds it necessary. For instance:

U: an evaluation is an annotation with a grade
A: what is grade? (type or attribute)
U: attribute
A: what are the possible values of "grade"?
U: 1,2,3,4,5
A: are 1,2,3,4,5 disjoint values?
U: yes

Let us now consider a last example, extending an existing operation annotate (document):

\begin{verbatim}
annotate (document)
before: exists (document)
after: exists (document),
    new instance: (annotation),
    new relation: (document) annotated-by (annotation)

U: to profile is like to annotate a user
A: to profile (user)
A: before: exists (user)
\end{verbatim}
Our approach brings about the need for a tool to support design, in which the designer can represent his model and run simulations of possible extensions to it. The tool should also provide for special tags and rules that govern how extensions are made, in which order the elements in the model are traversed when generating possible interpretations, incompatibility tags between pairs of elements, cardinality of attributes, and the like. A decision the designer must make is as to whether he should make this kind of control and rule customization available to users, and to what extent.

6. Discussion

Our ultimate goal is that every linguistic construct afforded by the user interface language be interpreted. Of course we suppose that users follow the Gricean maxims [17] while interacting with an extensible artifact of the kind we propose. In other words, they do not intentionally introduce falsehoods in conversation, neither do they choose to be intentionally inefficient in expressing themselves. So, based on such premises, we attempt to maximize the usage of interactive patterns.

Users do not need to have a “formal” knowledge about the underlying application model. They need, however, to be able to express themselves through the user interface language. From the user’s point-of-view, the application model is the semantic model of the whole range of discourse he can produce. In our approach, if a user’s utterance does not have a literal, syntactically correct and semantically precise meaning, our extension mechanisms take over and do one of the following, depending on the kind of utterance: (1) try to provide an interpretation for that utterance, or alternative interpretations for the user to disambiguate, taking into account the existing model (Searle’s world-to-word direction of fitness [31]); or (2) try to modify the semantic model to fit the corresponding utterance (Searle’s word-to-world direction of fitness).

It is important to emphasize the cultural-dependent nature of our extension mechanisms. Different cultures have different ways of understanding and using metaphors and metonymies, even if they share a common language. Our own native language, Portuguese, is a good example for such differences, since European Portuguese is observably less tolerant to the use of metonymies, for example, than Brazilian Portuguese.

Since we are resorting to a subset of natural language to allow users to extend software, a great challenge is to choose adequate lexical signs, without which it would be difficult or even impossible to produce sound metaphorical or metonymic utterances. For instance, we may need to define words or expressions that represent the inverse of a relation, such as “wrote” as an inverse of “written-by”.

In order to produce usable EUP environments that make use of our extension mechanisms, the designer must take great care in representing the application model in a KB. As future work, we intend to provide a tool that helps designers fine-tune their KB, so as to maximize the gain of allowing for metaphorical and metonymic extensions. The features of such a tool include a generative mechanism that generates potential non-literal utterances and their corresponding literal interpretations; and an analogy-recognition mechanism that navigates through the KB in search of seemingly analogical relations, so as to make this explicit and uncover potential inconsistencies or opportunities for adjustments.

We have described a powerful interpretive mechanism that can handle non-literal expressions. When incorporated in an extensible application’s interface, they allow users to express themselves inaccurately, by means of a metaphorical or metonymic utterance that refers to known elements of the domain. Following a process of abductive reasoning, our mechanism generates literal interpretations to the metaphorical or metonymic formulations. The resulting interpretation has a vanishing nature, i.e. the substitution is only considered valid for the current situation and context. Permanent changes must be explicitly intended by users.

Interfaces built according to this approach have enhanced interpretive intelligence, in that they are able to assign meanings by generating abductive interpretations. They also exhibit operational intelligence, since they can actually act upon the application model and produce modifications equivalent to end-user programmable ones, proposed by macro recording, scripting or programming by demonstration approaches.

Our main contribution for intelligent user interfaces is that, with metaphorical and metonymic operators, our approach considerably increases the users’ communicative power, and allows them to follow a more “natural” pattern of thought and expression. Thus, it increases the users’ chance of achieving their goals, even when they do not have a complete model of the application and underlying domain. Another related benefit arises from the fact that our mechanism can generate not only possible interpretations, but also explanations about these interpretations, if reasoning paths are unfolded. So, it can teach users about the underlying models and design assumptions. Additionally, given the power of metaphors and metonymies in our
acquisition of new knowledge and information [21,23,27], this may potentially reduce an application’s learning curve.

Because it is possible to generate multiple interpretations for a single non-literal utterance, we need heuristics to disambiguate and give precedence to interpretations. Our model should be applied to a variety of domains, in order to produce more refined heuristics. We believe that some universal heuristics may be found across domains, but that more sophisticated heuristics may prove to be domain-dependent. We should then decide whether the latter should be embedded in the interpretive mechanism, or disambiguation should be left to users at runtime.

Our future work involves fully integrating our model into the existing collective knowledge organization system. It is likely impractical to model a complex application in its entirety, so that its potential metaphors and metonymies can be consistently handled. Nevertheless, it may be possible to model a subset of the application in more detail, taking our interpretive and extension mechanisms into account. The designers would have to try to anticipate users’ needs for enhanced interpretive abilities, or for extensibility, and calibrate the specificity of the model accordingly.

We acknowledge the overhead on the designers as more decisions and more unknown variables come into the picture. Nevertheless, we plan to provide them with tools to facilitate the modeling process, and to help anticipate potential distortions that may arise from the application of our mechanisms to the intended model. Such design tools should not only guide designers through the representation process, but also generate a set of metaphorical utterances corresponding to each literal utterance. Thus, the designer would be able to use this information to fine-tune representations and correctly express his assumptions about the domain and the application.

Acknowledgements

The authors would like to thank CNPq, FAPERJ and Tecgraf for providing financial support to this work. They would also like to thank the Semiotic Engineering Research Group at PUC-Rio for their contributions to some of the ideas presented in this paper.

References