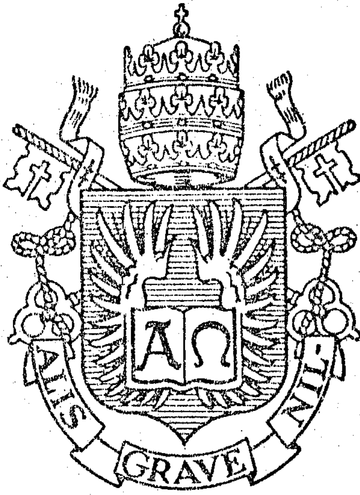


PUC

Series: Monografias em Ciência da Computação
No.14/88

AN APPROACH TO SOFTWARE REUSE BASED ON A FORMAL THEORY OF METAPHORS

Carlos J. P. Lucena
José Reinaldo Silva

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
MARQUÊS DE SÃO VICENTE, 225 — CEP 22453
RIO DE JANEIRO — BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação, No.14/88
Editor: Paulo A. S. Veloso Agosto, 1988

AN APPROACH TO SOFTWARE REUSE BASED ON A FORMAL THEORY OF METAPHORS *

Carlos J. P. Lucena
José Reinaldo Silva **

* Partially supported by ILAT/IBM Brazil, CNPq and FINEP
** Universidade de São Paulo, SP, Brazil

In charge of publications:
Rosane Teles Lins Castilho
PUC/RJ - Departamento de Informática,
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Resumo

Este artigo apresenta um estudo sobre a relação existente entre reutilizacão de software e a formalizacão da teoria de metáforas e analogias. Mais especificamente, a formalização de teoria de metáforas é apresentada como uma metáfora para o problema de reutilização de software. São considerados dois domínios: o domínio fonte e um domínio alvo. O mapeamento de símbolos entre estes dois domínios é o passo inicial para a transferência de informação através de metáforas. É apresentado um exemplo que demonstra a viabilidade do paradigma de reutilização baseado em metáforas, quando uma metodologia sistemática de desenvolvimento é aplicada tanto ao domínio fonte quanto ao domínio alvo. Finalmente, os aspectos computacionais do método são apresentados enfatizando a possibilidade de criação de uma ferramenta de reutilização com base no paradigma proposto.

Palavras - chave: Reutilização de software, metáforas, analogias, restrições de estrutura, metodologia de desenvolvimento, especificações formais.

Abstract

The present paper examines the relationship between software reuse and a formal theory of metaphors and analogies. More specifically, we have used a formal theory of metaphors as a metaphor for the problem of software reusability. Metaphors use symbols belonging to a domain, called the source domain, to refer to objects that belong to a possibly different domain, called the target domain. We explore an example by demonstrating the viability of reuse when a systematic development method is applied in both the source and the target domains. We also examine the computational aspects of the method to show that it may become a promising technique for the reuse of software designs.

Key-words: software reuse, metaphors, analogies, structural constrainst, development method, formal especification.

An Approach to Software Reuse Based on a Formal Theory of Metaphors

Carlos J. P. Lucena (*)

Pontificia Universidade Catolica

Rio de Janeiro - Brazil

José Reinaldo Silva (**)

Unversidade de São Paulo

São Pauló - Brazil

Abstract

The present paper examine the relationship between software reuse and a formal theory of metaphors and analogies. More specifically, we have used a formal theory of metaphors [5] as a metaphor for the problem of software reusability. Metaphors use symbols belonging to a domain, called the source domain, to refer to objects that belong to a possibly different domain, called the target domain. We explore an example by demonstratinhg the viability of reuse when a systematic development method is applied in both the source and the target domains. We also examine the computational aspects of the method to show that it may become a promising technique for the reuse of software designs.

key words : reuse, metaphors, analogies, structural constrainsts, development method, formal specification

1. Introduction

Software reuse is believed to be a key to improving software development productivity and quality. Many approaches to reuse have been studied. (See below for some references to this work.) Still, there is a lack of proper foundations for addressing the issues involved. We propose in this paper an approach to reuse which provides both an explanation of the underlying concepts as well as some feasible mechanisms to support the process. The approach is based on a formal theory of metaphors (and analogies).

There are two levels of reuse to consider : the reuse of ideas and knowledge, and the reuse of particular artifacts and components [1]. As Biggerstaff and Richter put it in [2], this is the question of code reuse versus design reuse. Code reuse pays off when the characteristics of the problem domain allow the establishment of standards in the domain and these standards lead to the success of reusability in the domain. That is, a very specific domain makes the reuse of code manageable. Methods such as "building blocks", and "subroutine libraries" and new approaches based on object oriented programming [3] explore these possibilities.

In most complex domains, code itself is not reusable. Nevertheless, general design ideas are often quite reusable. This observation has led to the notion of reusing design information. The problems with this idea have to do with how to represent the design information and how to associate this type of reuse to the specification and design process.

It has been noted in [4] that ideally the software development process should start by developing an intuitive understanding of the problem to be solved and examining the existing software or manual procedures designed to deal with similar problems. The originators of this observation claim that formal specifications can be used as a design tool [4]. In the present paper we try to go one step further, by

* Partially supported by Instituto de Enhenharia de Software/ILAT, IBM Brazil

** Partially supported by CNPq and IBM Brazil

trying to outline a formal notion of reuse that can be incorporated into the formal specification and design process.

When discussing research issues, Biggerstaff and Richter [2] point out that the fundamental problem preventing the successful reuse of design information is finding a representation of that design information which allows designs to be captured in a form that is readily machine-processible. They believe that the central mechanism for solving the problem of reusing design information is the notion of "semantic binding" or, as they call it : "binding by analogy". According to the authors "applying a design from one context to a new and different context, will provide the most general form of reuse". Freeman had expressed the same idea by stating that reuse is the use of previously acquired concepts and objects in a new situation [1].

Since we share the ideas expressed above, we have decided to investigate the relationship between software reuse and some theories of metaphors and analogies. More specifically, we have used a formal theory of metaphors [5] as a metaphor for the problem of software reusability. In this paper we use Indurkhya's theory [5] of metaphors almost without change and try to convince the reader through exemplification and a brief study of some computational aspects that it sheds a considerable light (to use a metaphor) on the problem of reusability.

Carbonell [6] considers metaphors, similes and analogies as just different linguistic manifestations of the same underlying cognitive process : analogical reasoning. A metaphor consists of three parts : target, source and analogical mapping. Analogical problem solving can be summarized as a four-stage process [6]:

(i) recallling one or more past problems that bear strong similarities to the new problem;

(ii) constructing a mapping from the old problem solution process into a solution process for the new problem situation;

(iii) instantiating, refining and testing the potential solution to the new problem;

(iv) generating recurring solution patterns into reusable plans for common types of problems.

Items (i), (ii) and (iii) take place when we are dealing with design reuse. Code reuse deals with objects produced in phase (iv).

Indurkhya's theory [5] formally explores the following characteristics of metaphors :

(i) metaphors use symbols belonging to a domain, called the source domain, to refer to objects that belong to a possibly different domain called the target domain;

(ii) the target and source domains need not be different or disjoint; they can be the same or partially overlap;

(iii) a metaphor works by coherently transferring a set of structural constraints from the source to the target domain;

(iv) the adequacy of a metaphor is determined, at least in part, by the amount of structure that is coherently transferred from the source to the target domain.

This theory has been called constrained semantic transference. In the next section we will present the essential parts of this theory (from our point of view), while establishing some parallels between aspects of the theory and the reusability problem. Later, we develop an example of the application of the theory to software design reusability. We explore the example by demonstrating the viability of reuse when a systematic development method is applied in both the source and the target domains (ideally it should be a formal method). Finally, we explore the computational aspects of the method to show that it may become a promising technique for the reuse of software designs.

2.Central Concepts in Indurkhya's Formal Theory of Metaphors

In this section we present the basic definitions and a theorem that constitute the essence of Indurkhya's formal theory of metaphors [5]. As we have anticipated in the introduction, we will use this theory as a metaphor for the software reusability problem. As we introduce the concepts in the theory, we establish an informal association between them and some key issues in the area of reusability. In the following section we illustrate these ideas through an example and indicate how we are presently exploring the ideas we first expose in this paper.

We consider a set V of tokens defining a given language over a set of types $\Omega = \{o_0, o_1, \ldots, p_0, p_1, \ldots\}$ where type o_n stands for an n-ary operations and type p_n is a predicate of arity n. We now define a function $f: V \rightarrow \Omega$ that associates a type to each token in V. A vocabulary is a pair, $<V,f>$. A simple example of a vocabulary can be expressed as follows :

$$V = \{ \text{node, root, edge, leaf} \}$$

$$f(x) = \begin{cases} p_1 & \text{if } x = \text{node, root, leaf} \\ p_2 & \text{if } x = \text{edge} \end{cases}$$

where node, root, edge and leaf have the usual graph theoretical meaning.

Def] A domain is defined as a quadruple $< V, f, S, S_d >$ where the pair, $< V, f >$ represents the vocabulary over which it is defined. S is a consistent set of sentences over this vocabulary, called structural constraints since they restrict the semantic interpretations of the sentences that can be formed with the vocabulary $< V, f >$ and S_d is a subset of

S whose elements are derivations (definitions) of tokens in $<V,f>$.

In the domain of oriented trees, the sentence

$$\forall x\ [\text{leaf}(x) \Leftrightarrow \forall y\ [\text{node}(y) \Rightarrow \neg\text{edge}(x,y)]]$$

should belong to S, since it restricts the possible interpretations of the token leaf to the cases in which there are no edges starting at x. On the other hand, the same sentence is a definition of "leaf" and therefore it belongs to S_d.

A mapping between two domains $D_1 = <V_1, f_1, S_1, S_{d1}>$ and $D_2 = <V_2, f_2, S_2, S_{d2}>$ is a partial function $F: V_1 \rightarrow V_2$. Therefore, if the domain D_1 is the domain of directed trees and D_2 the domain of family relations, a mapping between these two domains could promote the semantic transference between a node person, in which a node in a directed graph is associated, with a person in a family tree. It is necessary that node and person have the same type, that is, $f_1(\text{node}) \doteq f_2(\text{person}) = p_1$. The preservation of type p_1 is essential to ensure the coherence of the metaphor.

Def] A mapping F between the domains $D_1 = <V_1, f_1, S_1, S_{d1}>$ and $D_2 = <V_2, f_2, S_2, S_{d2}>$ is said to be *admissible* if for every $v \in V1$, if $v \in \text{Dom}(F)$ then $f_1(v) = f_2(F(v))$.

By the above definition, any set of sentences S, $S \subseteq S_1$, can be transformed into a new set of sentences provided that each constant, function and predicate in S belongs to V_2 or belongs to an admissible mapping F from D_1 to D_2. (This is achieved by extending F as the identity to symbols in S belonging to V_2.) Transformability is not a sufficient condition to guarantee the coherence of a metaphor.

In the first place, a metaphor $T = <F, S>$ is characterized by an admissible mapping and by a transformable set of sentences S of the source domain that we want to project to the target domain.

Def] A metaphor $T = <F, S>$ between the domains $D_1 = <V_1, f_1, S_1, S_{d1}>$ and $D_2 = <V_2, f_2, S_2, S_{d2}>$ is said to be coherent if the set of sentences $S' = F(S) \cup S_2$ is consistent.

Since the notion of consistency is not decidable, a software reutilization tool based on metaphors must use alternative methods to verify the coherence of the executed semantic transferences. We will come back to this point later.

An interesting special case of a coherent metaphor that is very useful in the case of reutilization occurs when every sentence in $F(S)$ is a logical consequence of S_2. In this case the metaphor is said to be strongly coherent.

Def] Let $S \subseteq S_1$ be a set of sentences transformable by an admissible mapping F. We say that S is maximal with respect to F if every sentence in S_1 that is transformable by F is in the closure of S.

A metaphor $T = <F, S>$ is said to be maximal coherent or maximal strongly coherent if S is maximal and T is coherent or strongly coherent respectively. That concept will be used latter on in the definition of a "useful metaphor".

Finally, we emphasize the notion of invertibility of a metaphor. It will be used in reutilization to allow the user to incorporate a description of a component into the source domain, whenever a description (specification) of a reusable component is identified.

Theorem] Let $T = <F, S>$ be a metaphor in which F is an injective mapping between the domains D_1 and D_2. Then, there exists an inverse mapping F^{-1} from D_2 to D_1 and an inverse metaphor $T^{-1} = <F^{-1}, S'>$, (where $S' \supset F(S)$), which is coherent if T is so. (The proof that invertibility preserves coherence is based on Craig's Interpolation Lemma [7]).

In reusability terms, the coherence of the inverse metaphor indicates the possibility of using reusable modules to support the development of a software specification. It is clearly the case when a reusable component is characterized for a set of sentences $S' \supset F(S)$. The inverse metaphor is then used to map the sentences from $S' - F(S)$ to the source domain.

On the other hand, if the metaphor is strongly coherent, it is not necessarily the case that the inverse metaphor is. That is, although by the above theorem coherence is preserved by inversion, the same does not happen in the special case of strongly coherent metaphors. For example, there can exist a symbol p in the target domain such that $F^{-1}(p)$ is not defined in S_1, and the possibility of reusing specifications of S' is conditioned to the ability of enlarging the source domain.

Although we do not explore fully in this paper the idea of reusing specifications of the target domain to construct the source, we will show how operations defined over metaphors, such as augmentation, can play this role in reutilization.

3. Operations over Metaphors

Given a coherent metaphor T $= <F, S>$ from D_1 to D_2, where $D_1 = <V_1, f_1, S_1, S_{d1}>$ and $D_2 = <V_2, f_2, S_2, S_{d2}>$, we take v_2 to be a constant belonging to V_2, d_2 to be its respective derivation (definition), which is a sentence belonging to S_{d2}. Let v_1 be a constant that does not belong to V_1. We assume that v_2 does not belong to the mapping F. If T is invertible the augmentation operation is such that

$$\text{Augmentation}(D_1, D_2, T^{-1}, v_2, d_2, v_1) = <T', D'_1>$$

where $D'_1 = <V'_1, f'_1, S'_1, S'_{d1}>$ is the new augmented source domain, that is :

$V'_1 = V_1 \cup \{v_1\}$;
$f'_1 = f_1$ augmented by the relation $f'_1(v_1) = f_2(v_2)$, which preserves the admissibility of mapping F;

$S'_1 = S_1$ augmented by the sentence $F'^{-1}(d_2)$;

$S'_{d1} = S_{d1}$ augmented by the derivation $F'^{-1}(d_2)$

$T'^{-1} = \langle F'^{-1}, S'' \rangle$ is a new metaphor where $S'' = S' \cup \{d_2\}$, and

$F'^{-1} = F^{-1}$ augmented by the mapping $\{v_2 \rightarrow v_1\}$ which preserves inversibility and coherence.

Augmentation allows the inclusion of definitions in the source domain based on the derivations in the target domain. This is the simplest way by which the approach to reusability based on metaphors can be applied to help the specification of the source domain.

The above operation can be extended to cover the cases in which one wants to augment the source domain when the correspondent constants in the target domain do not have derivations. This extended operation can also transfer a hole structure from the target domain once it is consistent with $F^{-1}(S')$. Such an operation is called "positing structure" [5].

From the point of view of reutilization we assume that if we have in the source domain a set of specifications to be elaborated and as a target domain a set of descriptions (specifications) of components, and if there exists a coherent and invertible metaphor T between them, then using T^{-1} and the augmentation operation it is possible to extend the source domain, that is, to reuse specifications.

A computable version of the semantic transference method, called approximate semantic transference [8], is described in the next section.

4. A Computable theory of metaphors

The concept of coherence is based on the concept of consistency and, therefore, is not computable when applied to open end domains. That is, domains that may grow and therefore extend the structural constraints, as occurs in typical reutilization domains.

The method of approximate semantic transference [8] replaces the concept of consistency by the concept of w-consistency. This new notation is formulated in terms of the principle of resolution/refutation [9].

By the resolution theorem, if S is a set of clauses, S is unsatisfiable iff the $n^{th.}$ resolution in S contains the empty clause,

$$\square \in \Re^n(S)$$

for some n greater then or equal to zero, where the $n^{th.}$ resolution of S is given by,

$$\Re(S) = \begin{cases} S, \text{se } n = 0 \\ \Re^{n-1}(S) \cup \Re(\Re^{n-1}(S)), \text{if } n \geq 1 \end{cases}$$

The concept of w-coherency can now be defined as :

Def] A set of clauses S is said to be w-inconsistent if $\square \in \Re^w(S)$

S is w-consistent if it is not w-inconsistent. A clause X occupies a level n with respect to S if n is the smallest natural number such that X belongs to the $n^{th.}$ resolution of S.

Finally, the concept of w-consistency for a given domain is defined in the following way :

Def] A domain is said to be w-consistent for a given natural number w if its set of structural constraints is w-consistent.

5. Software Reutilization Using Semantic Transference : A case study

In this section we shall examine a simple but realistic example of software reutilization using semantic transference.

We take as an example a situation in which the target domain is a description of a software component that we will call Help_Assistant. It consists of a set of procedures that, when called by the program, issue messages, warnings and/or menus and windows that help the user to find a desired command. In the simplified version that we will use, the Help_Assistant is able to access three types of procedures :

A (Error Procedure) : a procedure of type A is requested when a fatal error occurs in the host program; in this case the Help_Assistant notifies the fatal error to the user, issues a message and informs the host of the "returning point"; a procedure type A causes an interruption in the sequence of execution of the host program.

B (Warnings) : a procedure of type B is called when a non-fatal error occurs in the host or when it is necessary to communicate a message or warning during the execution of a sequence of instructions in the program that uses the Help_Assistant ; the host is not interrupted in this case.

C (Assistance to the user) : the call takes place when the user needs to find, through an interactive process, a given command or access procedure; the access to this type of procedure may or may not be originated as a consequence of a fatal error.

In our example the source domain is constituted by a description (yet incomplete) of a friendly interface to a

database. The interface has a main menu which can access windows ("forms") which, in turn, help the user to compose the queries to the DB. If an error occurs in the composition of a query or during an attempt to make an illegal access to the DB, the interface must issue error messages interrupting the sequence of execution. Each "form" is characterized by a type ("msg" or "form") depending on the action required : "msg" if an access error occurs and "form" if the system must display on the screen a window which is capable of accepting a command chosen by the user.

Before applying the formal theory of metaphors we will explore informally the association between Carbonell's [10] strategy for problem solving by analogy and Jackson's [11] well-known approach to program design.

According to [10] a search for solutions to problems similar to the one at hand must compare differences among the following :

(i) the initial state of the new problem (target) and the initial state of a previously solved problem (domain) ;

(ii) the final state of the new problem and the final state of a previously solved problem ;

(iii) the path constraint under which the new problem must be solved and the path constraint presented when the previous problem was solved ;

(iv) the proportion of operation pre-conditions of the retrieved operator sequence satisfied in the new problem situation (this is called the applicability of a candidate solution [10]).

Jackson [11] structures the input and output domains of a given program and looks for mappings between these two domains : the mappings are the program. So if both the target and the source domains for reusability were structured following Jackson's data flow approach we will be able to find the path constraints mentioned above which are the structural constraints of the metaphor theory.

In figure 1 below we indicate the structural constraints between the domains structured according to Jackson's methodology.

Note that the structural constraints do not only define the type of output as the diagram in Figure 1 seems to imply. They also express the conditions that generate the different outputs (path constraints), as will become clear later.

In the sequel we present the formal definitions of the source and target domains described above.

a)Domain D_1 (DB Interface)

Vocabulary V_1 = { buffer, pointer, command, parameters, input, output, Interface, Form_File, equal, msg, form }

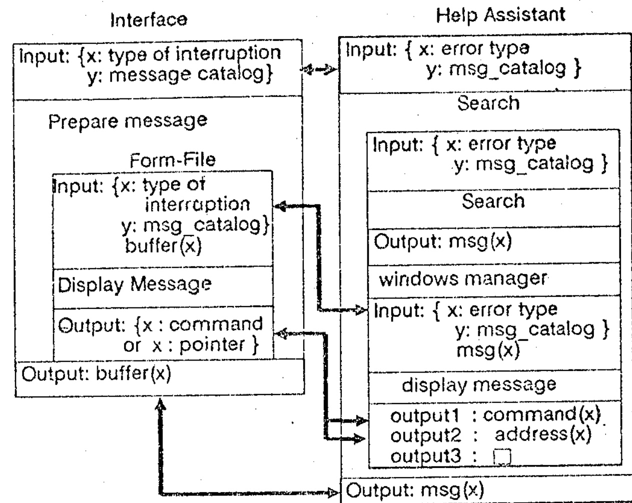


Figura 1

$$f_1(X) = o_0 \quad X = \text{Interface, Form_File, msg, form}$$
$$f_1(X) = o_2 \quad X = .$$
$$f_1(X) = p_1 \quad X = \text{buffer, command, pointer}$$
$$f_1(X) = p_2 \quad X = \text{output, equal, parameters}$$
$$f_1(X) = p_3 \quad X = \text{input}$$

where f_1 is the mapping from V_1 into { $o_0, o_1, o_2, \ldots, p_0, p_1, p_2, \ldots$ } and o_i is an operator of order i and p_i a predicate of arity i.

Structural Constraints :

$E_1^{(1)}$: $\forall x \, \forall y[\text{input}(x,y,\text{Interface}) \Leftrightarrow \text{parameters}(x,y)]$

$E_1^{(2)}$: $\forall x[\text{output}(x,\text{Interface}) \Leftrightarrow \text{buffer}(x)]$

$E_1^{(3)}$: $\forall x \, \forall y \, [\text{input}(x,y,\text{Interface.Form_File}) \Leftrightarrow [\text{input}(x,y, \text{Interface}) \wedge \text{buffer}(x)]]$

$E_1^{(4)}$: $\forall x[\text{output}(x,\text{Interface.Form_File}) \Rightarrow [\text{input}(y,z, \text{Interface}) \wedge \text{equal}(y,\text{msg}) \wedge [\text{buffer}(z) \Rightarrow \text{pointer}(x)]]$

$E_1^{(5)}$: $\forall x[\text{output}(x,\text{Interface.Form_File}) \Rightarrow [[\text{input}(y, z, \text{Interface}) \wedge \text{equal}(y,\text{form})] \Rightarrow [\text{command}(w) \wedge \text{equal}(x, w)]]]$;

Derivations $D_1^{(1)} = E_1^{(1)}$; $D_1^{(2)} = E_1^{(3)}$

where each derivation is a structural relation that "defines" a given symbol of the vocabulary $< V, F >$

b) Domain D_2 (Help_Assistant)

Vocabulary V_2 = { record, input, output, msg, Help_Assistant, Search, Window_Manager, command, equal, address, empty, A, B, C }

$f_2(X) = o_0 \quad X = $ Help_Assistant, Search, Window_Manager, A, B, C
$f_2(X) = o_1 \quad X = .$
$f_2(X) = p_1 \quad X = $ msg, command, address, empty
$f_2(X) = p_2 \quad X = $ output, record, equal
$f_2(X) = p_3 \quad X = $ input

Structural Constraints :

$E_2^{(1)}$: $\forall x\, \forall y$[input(x,y,Help_Assistant) $\Leftrightarrow$ record(x,y)]

$E_2^{(2)}$: $\forall x$ [output(x,Help_Assistant) $\Rightarrow$ msg(x)]

$E_2^{(3)}$: $\forall x\, \forall y$[input(x,y,Help_Assistant.Search) $\Leftrightarrow$ $\Leftrightarrow$ input(x, y, Help_Assistant)]

$E_2^{(4)}$: $\forall x$ [output(x,Help_Assistant.Search) $\Rightarrow$ msg(x)]

$E_2^{(5)}$: $\forall x\, \forall y$ [input(x,y,Help_Assistant.Window_Manager) $\Leftrightarrow$ [input(y, z, Help_Assistant)] $\wedge$ msg(x)]]

$E_2^{(6)}$: $\forall x$ [output(x,y,Help_Assistant.Window_Manager) $\Rightarrow$ [[input(y,z,Help_Assistant)] $\wedge$ equal(y, C)] $\Rightarrow$ [command(w) $\wedge$ equal(x, w)]]]

$E_2^{(7)}$: $\forall x$[output(x,y,Help_Assistant.Window_Manager) $\Rightarrow$ [[input(y,z,Help_Assistant)] $\wedge$ equal(y, A)] $\Rightarrow$ address(x)]]

$E_2^{(8)}$: $\forall x$[output(x,y,Help_Assistant.Window_Manager) $\Rightarrow$ [[input(y,z,Help_Assistant)] $\wedge$ equal(y, B)] $\Rightarrow$ empty(x)]]

Derivations $D_2^{(1)} = E_2^{(1)}$; $D_2^{(2)} = E_2^{(3)}$; $D_2^{(3)} = E_2^{(5)}$

In the present example the structure of the software components was found by following the data flow of the sub-components. Of course the formal theory of metaphors, when used as a metaphor for reutilization, is not dependent on the reutilization method to be used.

The choice of the data flow approach (Jackson's method [11]) serves two purposes:

(i) it is a simple development method and it does not produces a long list of structural constraints;

(ii) it offers us an opportunity to show one example in which the two vocabularies (source and target) have symbols in common without being reducible to the same vocabulary.

With respect to (ii) it is important to note that if $V_1 = V_2$ there is only one chance of finding a metaphor $T = <F, S>$: to map all sentences belonging to a set $S \subseteq S_1$ identically to corresponding sentences in S_2. This implies that we will find a reusable component only when the given structured constraints are a perfect matching of S_1 by the metaphor T. In other words, when we have the situation "use-as-is".

In the example, the symbols msg, input, output, command and equal are common in the two domains. We can therefore choose the metaphor $T = <F', S>$ where F' is the mapping:

{(buffer→msg),(Interface→Help_Assistant), (Form_File→Window_Manager),(parameters → record),(pointer→address),(form → C),(msg → A)}

Note that the symbol msg $\in V_1$ is a constant (arity zero) while the symbol msg $\in V_2$ is a predicate of arity one and, therefore, they are not common symbols.

F' is clearly admissible and the metaphor $T = <F', S>$, where S is the set of all structural constraints of the domain D_1, gives us a set of sentences S'_2 of the target domain where

$$S'_2 = \{ E_2^{(1)}, E_2^{(2)}, E_2^{(5)}, E_2^{(6)}, E_2^{(7)} \}$$

This set of sentences defines the data flow that enters Help_Assistant and flows through the component Window_Manager. It suggests that in the specification of the interface for the database we may reuse the Window_Manager component.

The most serious problem is to eliminate the inconsistent mappings, such as :

$$[F' - \{(\text{msg} \rightarrow \text{A})\}] \cup \{ (\text{msg} \rightarrow \text{msg}) \}$$

which is not admissible, or

F' = {(buffer→msg),(Interface→Window_Manager), (msg→ C),(form → A),(Form_File → Help_Assistant), (parameter→record) }

which may lead to a conclusion that there is no reusable component. The adequate selection of mappings requires an extensive use of the heuristics extracted from the development methodology selected. Also a crucial issue is how to associate the process of generating the structural constraints with a given development methodology. In other words, it is necessary to find a way to transform specifications into formal relations. We are approaching this last problem by restating Indurkhya's formal theory of metaphors in terms of interpretations between theories [12], a notion that also allows us to talk formally about the process of formal development.

For the purposes of this paper we will use Indurkhya's metaphor theory as it is and present in the next section a discussion of the computational aspects of directly applying it to software reutilization. Even if we drop this approach in the future because, for instance, the changes we introduce in the formalism suggest better algorithms, we think that some exploration of the computational aspects helps in comprehending the role of metaphors in reutilization.

6. Computational Aspects of the Semantic Transference Approach for Reusability : The Search for Adequate Development Methodologies.

A computational strategy directly derived from the formalization of metaphors that we have used leads to unreasonable algorithms from the efficiency point of view. For example, if the vocabulary of the source domain has N symbols and the vocabulary of the target domain M symbols $(M > N)$ there exists $M!/(M - N)!$ possible mappings leading to an algorithm which is $O(M^N)$.

Therefore it is fundamental for the implementation of the metaphor paradigm, "to find a mapping that takes us to a solution", to be able to recognize "useful metaphors". If we are able to identify the properties that the useful mappings must satisfy, they can be expressed as "elimination rules" which can reduce considerably the search effort.

One of these elimination rules can be found in the formalization of metaphors : the notion of admissible metaphor. If we call m_i and n_i the number of symbols of type *i* of the source and target domains, respectively, $(m_i > n_i)$, since

$$\Sigma m_i = M$$
$$\Sigma n_i = N$$

we have $\Pi (mi!/(mi - ni)!) \leq M!/(M - N)!$

Even with the above improvement in the algorithm we are talking about infeasible run-times. This is especially true if we are dealing with reuse at the level of programming-in-the-large [13] (systems specification and design) where a very large number of symbols are expected in both source and target domains.

In what follows we are going to explore some general alternatives based on the notion of useful metaphors that incorporate the general characteristics of programs and their specifications.

We have seen that a metaphor T is defined as a pair $<F, S>$ where F is an admissible mapping and S is a subset of the source domain. The result of the application of a metaphor is a sub-structure $S' = F(S)$ of the target domain (assuming that $S_2 \cup F(S)$ is consistent, that is, a coherent metaphor or rather a w-coherent metaphor).

We will say that $T = <F, S>$ is a useful metaphor if F is an admissible mapping and S is a maximal consistent subset of axioms of the source domain. We use the set of derivations as an approximation for this maximal consistent set of axioms.

With respect to the mapping F we shall add to the notion of admissibility a classification of the symbols in the source domain. In the following discussion we will use the following classes : data, control and data transformers.

Some of these classes can be further subdivided into sub-classes as follows

(i) data	:	type (DT) structure (DS)
(ii) control	:	C
(iii) transformers	:	algorithms (AT) routines (RT)

Note that we have associated a mnemonic abbreviation to each sub-class. In this way the symbols may represent the specification of a given data structure (abstractly defined by one or various derivations) or a data type. Commands and control structures such as "command(*x*)" are characterized by the mnemonic C:

```
class_programS(command,C).
class_programT(command,C).
```

where the above "facts" establish that the symbol "command" of the source domain and the symbol "command" of the target domain are both control structures. Therefore, following the heuristic which says that equal symbols must mean the same thing, that is, must be mapped one into the other if they satisfy the admissibility criteria and the classification above, the element (command → command) should be selected first in forming the mapping F. The same applies to the transformers AT (algorithms) and RT (routine names).

Another possibility is to make a "good choice" of the sub-structures S to find a useful mapping. This implies finding structural constraints of the source domain that constitute a minimum specification set for the determination of the reusable component. In other words, if B is a subset of a set S'_i of the target domain component specification then, if S matches B,

$$\forall r \in S, F(r) \in B$$

a good candidate to be the S set is the set of all derivations of the source domain.

The derivations are structural constraints that define the symbols of the vocabulary V.

If there exists a reusable component there must also exist a set S'_k of derivations in S_2 such that $F^{-1}(S'_k) = S$, where $S = S_{d1}$ if we have a perfect match.

The problem is then to find a subset S of S_{d1}. In the worst case there are as many derivations as symbols in the source domain. Therefore we have at most 2^N candidates where N is the number of symbols of the source domain. Although this is still a prohibitive algorithm, the exponential time represents an improvement with respect to the factorial time we had before.

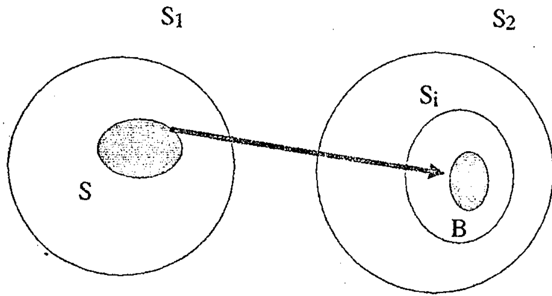


Figura 2

We shall now investigate a "reasonable" algorithm that explores at the same time:

(i) the notion of useful metaphor and admissibility
(ii) the general classification proposed for the "programming elements"
(iii) heuristics, such as : the same development method was used in both domains and equal tokens with the same arity must have the same meaning

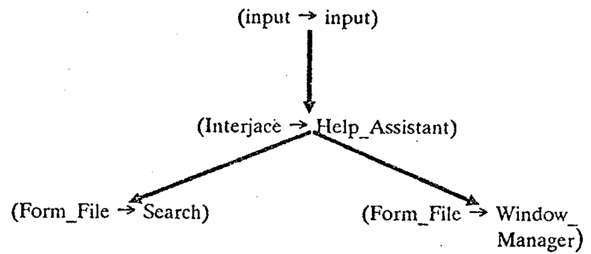
There are clear indications that it is possible to develop an algorithm (eventually a polynomial algorithm) that, starting from the derivations, maps the tokens involved in these sentences. Having this set of mappings at hand it proceeds by extending them to include other sentences. Following a selection criterion we can then abandon the developments that cover fewer tokens (something like the application of a branch-and-bound technique over a forest where each root is a derivation).

Example : Continuing our example of a metaphor suppose we have as set of derivations,

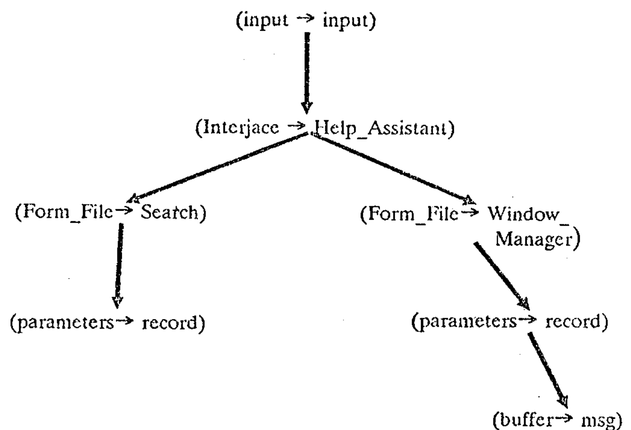
$$S = \{ D_1^{(1)}, D_1^{(2)} \}$$

From $D_1^{(1)}$ we have : (input → input) using the above mentioned heuristics. At the same time we can start a second tree $D_1^{(2)}$ whose mapping elements will be (input → input). Proceeding with $D_1^{(1)}$ it follows that the next symbol will be Interface, but in D2 the mapping possibilities would be $E_2^{(1)}$, $E_2^{(3)}$ and $E_2^{(5)}$.

Among those only $E_2^{(1)}$ has as its arguments a routine name which is not a composite one. The tree will then look as follows :



where the trees are generated by $D_1^{(2)}$. In the next step the mapping (parameters → record) must be chosen through the heuristics (as the only possibility) and added to the all other alternatives. With respect to $D_1^{(3)}$ the next token is Interface (already mapped) which will confirm $E^{(3)}$ and $E_2^{(5)}$ as true possibilities. As a following step the token will be buffer which eliminates $E_2^{(3)}$ as a possibility (given that $E_2^{(3)}$ has no more tokens to be mapped) and leaves as the only possibility (buffer →msg). The final tree is shown as follows



where the chosen path has the highest depth in the tree.

Following this mapping process we have :

F' = {(input→input),(Interface→Help_Assistant), (Form_File→Window_Manager), (parameter→record), (buffer→msg)}

which represent 57% of the total(*) mappings, besides having selected the mapping

$$D_1^{(1)} \rightarrow E_2^{(1)}$$
$$D_1^{(3)} \rightarrow E_2^{(5)}$$

7. Conclusions

We have suggested through exemplification and some algorithm schemas that a formal theory of metaphors seems to be a useful metaphor for the problem of software reutilization. It also seems that formal specification methods and a formal theory of metaphors can be used together to handle the problem of reuse of designs : apparently the most critical one in software engineering. For this approach to be entirely successful it is necessary to unify the theories supporting specification and the use of metaphors applied to software modules.

We think that the notion of interpretation between theories [12] can be used in this connection. It can first of all be used to put Indurkhya's theory of metaphors on a sounder and more general footing, providing a conceptually better basis on which to build a theory of reuse. This proof-theoretic basis can support more directly the development of appropriate heuristics and algorithms of the kind illustrated above. We are already well on the way to exploring this reformulation.

(*) The total number mappings generated by D1(1) and D1(3) are about 50% of the total number of mappings required to generate the best solutions (use-as-is type of metaphors).

[1] Prieto-Diaz, R. and Freeman, P. ; 'Classifying Software for Reusability', IEEE Software, January 1987

[2] Biggerstaff, T. and Richter, C. ; 'Reusability Framework, Assessment and Directions', IEEE Software, March 1987

[3] Meyer, B. ; 'Reusability : The Case for Object-Oriented Design', IEEE Software, March 1987

[4] Guttag, J. and Horning, J. J. ; 'Formal Specification as a Design Tool', in Software Specification Techniques, International Computer Science Series, Addison-Wesley, 1986

[5] Indurkhya, B. ; 'Constrained Semantic Transference: A Formal Theory of Metaphors', Synthese, 68, 1986

[6] Carbonell, J. G. and Minton, S. ; 'Metaphor and Common-Sense Reasoning', CMU-CS-83-110, Carnegie Mellon University, 1983

[7] Chang, C. C. and Keisler, H. J. ; 'Model Theory', North-Holland Publishing Co., 1978

[8] Indurkhya, B. ; 'Approximate Semantic Transference : A Computational Theory of Metaphors and Analogies', Cognitive Science 11, (1987)

[9] Robinson, J. A. ; 'A Machine-Oriented Logic Based on the Resolution Principle', Journal of ACM, vol 12, 1, 1965

[10] Carbonell, J. G. ; 'Learning by Analogy : Formulating and Generalizing Plans from Past Experience', CMU-CS-82-126, Carnegie-Mellon University, 1982

[11] Jackson, M. ; 'Principles of Programming Design', Academic Press, London, 1975

[12] Turski, W. and Maibaum, T. S. E. ; 'The Specification of Computer Programs', Addison-Wesley, 1987

[13] DeRemer, F. and Kron, H. H. ; 'Programming-in-the-Large Versus Programming-in-the-Small', IEEE Trans. on Software Engineering, SE-2, 2, June 1976