Concepts + Relations = “Abstract Constructs”[*]

Nicolas Anquetil
COPPE – Universidade Federal do Rio de Janeiro
C.P. 68511, Cidade Universitaria
RJ, 21945-970, Brazil
(55) (21) 590-2552 x.334
nicolas@cos.ufrj.br

Abstract

The goal of Reverse Engineering is to create an abstract representation of a system, identifying the concepts it implements and the relations between them. Both kind of information (concepts and relationships) have been the subject of various studies, but there are very few works that actually consider them jointly. In this article, we propose a method trying to remedy this deficiency.

We will present some experiments we performed on the Mosaic system and discuss their results. They show that our method is successful and can actually extract significant conceptual information. Examples of discovery of subsystem wrapping or extraction of concepts inheritance hierarchy are presented.

1 Introduction

Reverse Engineering is defined as “the process of analyzing a subject system with two goals in mind: (1) to identify the system’s components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction” [22]. Creating an abstract representation of a system implies discovering what abstract concepts it implements and what relations hold between these concepts. The initial efforts concentrated on finding abstract concepts, and results in this domain are promising. Extraction of relations between abstract concepts received less attention and does not extend beyond the classical implementation relations: inheritance and composition. But very little work, actually considered the extraction of concepts and relations to form abstract constructs.

Another important problem in reverse engineering is the “concept assignment problem” [6], or how to relate abstract concepts with portions of the code that implement them. In this domain too, research is progressing and solutions are starting to appear (e.g. [4]).

In this article, we describe an experiment to extract “abstract constructs” from a legacy software. We define these constructs as being composed of abstract concepts and relations between them. The concepts are directly connected to software components in the code and the relations between the concepts also come from the code. For example, studying the user interface of a software, we discovered a small inheritance hierarchy of widgets.

The organization of the article is the following: In section 2 we discuss the *state of the art* in concept and relation extraction and present the *related works*. Then, in section 3, we define more precisely the notion of *abstract construct* and present a first algorithm to extract examples of it. In section 4, we detail the *experiments* we conducted to find these constructs and we comment their *results* in section 5. Before concluding, we discuss some aspect of this research, limitations and future works.

2 Conceptual Information for Reverse Engineering

We propose, in this article, a method to extract abstract concepts and relations between them from a legacy software. Some methods already exist in this area and we will present them, highlighting the differences between these approaches and our. The discussion is decomposed in two parts, first we discuss extraction of abstract concepts, a topic which was well studied by others, then we come to the much newer problem of extracting conceptual relations.

2.1 Concepts

A fundamental goal of Reverse Engineering is to extract from the code abstract concepts that are significant to the

[*]This work is sponsored by a grant from the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ)

software engineers. There are a number of ways to do so depending on the source of information used:

Clustering: By grouping related things together, one defines concepts "in extension", that is to say, by the list of their members. Intuitively, the larger the cluster, the more abstract the underlying concept.

Software Components: Each and every software component can be said to implement a concept. One can assume that the more complex a software component is (e.g. structured type, long routine), the more abstract the related concept.

Clichés: A cliché is "a pattern describing salient features of a concept that supports recognition of that concept" [22]. Examples of clichés are: traversal of a linked-list, or handling of a counter. Recognizing cliché is therefore the action of looking for specific concepts in the source code. In theory, there is no limit to the level of abstraction of these concepts.

Documentation: Many words contained in external documentation, in comments or in identifiers directly relate to abstract concepts of the application domain or other knowledge domains. For example in the routine identifier "XtRemoveEventHandler", the word "event" refers to an abstract concept specific to event driven programming.

The *clustering* method can be based on informations from the code (references between software components) or on words from identifiers. In either case, the method leads to concepts that are difficult to understand because, to discover the "intention" (the semantics) of the concept, one must analyze the list of its members. This can only be done manually and with great difficulties. There are numerous examples of this method in the literature, for example [2, 17, 18].

Concepts extracted from *software components* are usually of a very low level of abstraction (e.g. the concept of a loop counter). Even complex components as structured types may implement low abstraction level concepts (e.g. a node structure to implement a linked list). There are also many examples of application of this method, more specially for extraction of classes from procedural code [9, 23]. These works are similar to what we propose (particularly for relations, see §2.2), but the concepts extracted are at a lower level of abstraction.

The method based on *cliché* recognition seems now receding due to scaling problems. It deals only with computer science concepts (see examples above), no application domain clichés has been defined to our knowledge and it is not clear what these could be. Another problem is that one has to know beforehand what concepts one will be looking for. We will see that our method does not have these limitations. Works in this area include [19, 21].

Words found in *documentation* are of mixed qualities, they may be utility words (the, a, is, ...), they may design concepts of importance or anecdotal, and finally they may be difficult to relate the source code. To solve the problem of utility words one can use a "stop list". Note that this problem can also be much alleviated by considering words in identifiers which are rarely adverbs or articles. To separate anecdotal concepts from important ones, one usually relies on the *repetition heuristic*: words which are more frequently repeated, have more chances to denote important concepts. An example of this is Sayyad [20], who extracted concepts from the comments of a legacy system. His approach requires some manual intervention mainly to correct errors due to the repetition heuristic (words which are frequent but do not denote important concepts). We avoid manual intervention because it is not compatible with the size of the systems usually dealt with in Reverse Engineering. Also Sayyad does not consider relations between the concepts.

Internal documentation (comments and identifiers) has the additional advantage of directly relating the abstract concepts (the words it contains) to the code (each identifier relates to a software component, comments can be related to the piece of code they describe). For example in [4], Antoniol *et al.* endeavor to recovering code to documentation links. Using frequent words in identifier and in sections of the external documentation, they link classes to parts of the documentation that describe them.

These two works do not extract relations between the concepts found.

2.2 Relations

The problem of finding conceptual relations in a legacy system has draw less attention, and the results are still primitive. For example most of the work only considers the basic relationships which are synonymy, inheritance and composition. Again, there are different approaches:

Clustering: Hierarchical clustering algorithms result in hierarchies of clusters, which specifies inheritance relation between the clusters.

Co-occurrence: If two concepts repeatedly appear jointly (e.g. the words "directory" and "file"), they have a good probability of being linked by a conceptual relation.

Documentation: By analyzing external documentation, one may be able to discover relations between concepts.

Implementation: It is well established that relationships like synonymy, inheritance or composition can be found between structured types from simple analysis of their definitions.

We already noted that the *clustering method* leads to concepts difficult to understand. Also, it is limited to the sole inheritance relation. This part includes also works on "concept analysis" [5] which, in this context, share the same problems as the traditional clustering method.

The method based on *co-occurrence* only allow to discover that there is a relation between two concepts, without specifying what this relation is. Also, it does not differentiate two related concepts (directory+file) from a nominal syntagms (data+base). The method is used in [15].

The method using *documentation* (e.g. [14]) could provide an improvement, but it requires complex analysis which is usually very difficult to perform automatically. The following two works [8, 12] simplify the problem by considering more structured documentation, Data Flow Diagrams, but they become dependent on the existence of this particular document which we judged over-restricting.

For lack of a better solution,, we chose to apply the method based on *implementation*. Its main problem is that it is currently limited to the three basic relations cited at the beginning of this section. The synonymy function has been much explored: detection of clone functions in [13, 16], detection of synonymous structured types in [7, 11, 11]. We say that a structured type *inherits* from another if it has the same fields plus some additional ones; we say that a structured type is *composed* of another if it has a field of the other type. Unfortunately, these two heuristics are often invalid. Inheritance can be implemented in procedural language using a pointer from the sub-type to the super-type, and in general, fields of another type (and pointers on another type) can be used to implement many different associations apart from composition.

3 Abstract Constructs

In this section we will set the basis of our method to extract concepts and relations between them from a legacy software system. For this, we present an introductory example and establish some basic definitions.

3.1 Basic Approach

The concepts will be extracted from the *documentation* and the relations from the code. The naive approach consists in:

α.*1* Extract concepts deemed important from the *internal documentation* (i.e. words in identifier) using the repetition heuristic

α.*2* Relate these concepts to software components

α.*3* Look for relations between these software components (method based on *implementation*)

α.*4* Deduce that these relations hold between the associated concepts

Table 1. First experiment, pairs of concepts "related" by the inheritance or composition relation number of occurrence of the pair for each relation.

Inheritance		Composition	
48	Event/Event	31	Event/Event
48	Event/X	31	X/Event
48	X/Event	36	Rec/Part
53	X/X	44	HT/HT
56	HT/Stream	62	Event/X
56	Stream/Stream	69	X/X
58	Stream/HT		
79	HT/HT		

However a first experiment with structured types of the Mosaic system leads to uninteresting results. Table 1 gives the most frequent triplet (concept1,relation,concept2) for the inheritance and composition relations. The only triplet which intuitively makes sense is {Rec,composition,Part}, which means that a rec(ord) is composed of a part. Studying this example a bit closer, we noted that the Mosaic system, and the X11 library on which it is based, contain several structures named xxxRec (ex: CompositeRec, ObjectRec, CompositeClassRec), all having a field member of type xxxPart (CompositePart, ObjectPart, CompositeClassPart). This organization is illustrated in Figure 1 (UML notation).

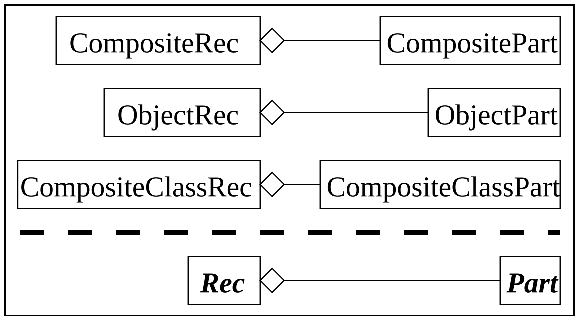


Figure 1. Example of abstract constructs found in Mosaic and the X11 library.

We propose that a reason of the failure of the first experiment is that it is looking for triplet in the entire system with almost no constraint. An experiment on relevance of identifiers to their definitions [1] showed that the names tend to be relevant "locally" and not for an entire system. We will not expand on the precise definition of this locality, but it seems normal that, in a legacy system, informal organization schemes (as naming convention) can only be followed locally. In the case of the rec(ord) being composed of a part, this locality boils down to the X11 library and the part of Mosaic directly related to it.

One could consider in this case that the locality corresponds to the definition of a subsystem. We felt that this locality could be difficult to use automatically and chosen to use another one. We refer here to the fact that the prefixes to "Rec" and "Part" in the related identifiers are the same. This seems a good marker of the organization scheme and it will be integrated in the algorithm (step α*.3*) as a constraint on what software components can be related (we will not consider those with very different identifiers).

3.2 The opposition heuristic

In step α*.1* of our algorithm, we applied the traditional *repetition heuristic*, that is to say the more frequent a word the more chances it has to denote an important concept. We propose to introduce a new heuristic:

Opposition heuristic: If two words are the only difference between two (long) sequences of words, they have good chances of denoting important concepts.

Consider, for example, the two identifiers XmRemoveEventHandler and XtRemoveEventHandler. Following the *repetition heuristic*, there is a good probability of the concepts Remove, Event and Handler being important. We propose that, following our *opposition heuristic*, there is a good probability of the concepts Xm and Xt being important too because they are the key features (concepts) that allow to differentiate the identifiers. Intuitively, concepts from the opposition heuristic will need less repetitions to be deemed important.

Note that if the opposition heuristic suppresses the need for repetition to identify the important concepts, it introduces the notion of a *context of importance* (or context of validity). The context of importance of the concept is defined by the software components it was extracted from. A possible way to characterize this context would be to consider the rest of the sequences of words from which the concepts were extracted with the opposition heuristic. In the example above, Xt and Xm would be important concepts in the context of {remove, event, handler} (what we called locality earlier). In practice, this context is more useful for the relations than for the concepts themselves. We found that the concepts extracted with the opposition heuristic were usually meaningful independently of their context. However, we will see that the relationships we will deduce between these concepts often make sense only in the specific context of importance of the concepts.

Eventually, if a concept is found in many different contexts, one will be able to deduce that it is important without restrictions, that is to say, the context of importance is the entire system. This is strictly equivalent to the repetition heuristic.

The naive algorithm we proposed remains the same, however, step α*.1* will be modified to use the opposition heuristic instead of the repetition one. The final algorithm we used is presented in section 4.

3.3 Definitions and Notation

To apply our opposition heuristic, we will need a tool to compute the *lexical difference* between two sequences of words.

Lexical difference (between two sequences of words) A set of *modifications* which, applied to the first sequence, would produce the second. There are three possible modifications: add(word), del(word), chg(word1,word2). For example a valid lexical difference between {the,blue,house} and {the,little,blue,cat} is {add(little),chg(house,cat)}.

Note that our definition of lexical difference does not specify where to apply the modifications, for example adding the word little before or after the pronoun. Consequently, there are potentially many valid lexical differences for two word sequences.

We will see later that there are cases where the difference between two sequences of words is that one word in the first is replaced by two words in the second (as "file" being replaced by "data+base"). In this case, the lexical difference will look like: {chg(file,data),add(base)}.

We will call the concepts we extract and the relations between them *abstract constructs*.

Abstract construct The association of a *lexical difference* between two identifiers (considered as word sequences) and a *relation* between the two associated software components.

In the Rec/Part example, the lexical difference between the two identifiers CompositeClassRec and CompositeClassPart is chg(Rec,Part):

$$\mathrm{Rec} \xrightarrow{change} \mathrm{Part}$$

The formal relation is the composition:

$$\mathrm{CompositeClassRec} \xrightarrow{composed} \mathrm{CompositeClassPart}$$

And the abstract construct is (see also Figure 1):

$$\text{Rec} \overset{composed}{\longrightarrow} \text{Part}$$

From now on, we will prefer the more compact notation: {chg(Rec,Part)}/composition, being understood that the first concept (Rec) is composed of the second one (Part). Similarly, in {chg(word1,word2)}/inheritance, it should be understood that the concept word2 inherits from the concept word1.

4 Description of the Experiments

We will now detail the experiment we did, to extract our abstract constructs from the Mosaic system. Mosaic is becoming one of the *de facto* benchmark system in the Reverse Engineering community. It is a medium-sized system ($\simeq$ 140KLOC of C code).

We used a different algorithm than the one specified in the previous section (although both are equivalent). Because we now use the opposition heuristic, we don't expect to get more than two "important" concepts per identifier pair (this was not true with the repetition heuristic). As a consequence, we found it easier to base the detailed algorithm on this notion of software component pair:

β.1 Select a set of *software components*

β.2 Decompose the software component identifiers in *word sequences*

β.3 Compute the *lexical difference* between pairs of software component identifiers

β.4 Find all the *relations* between pairs of software components

β.5 Find all related *pairs of software components* with a small lexical difference.

In step *β.1*, we experimented with two sets of software components, first with structured types and then with routines.

Decomposing the identifiers into sequences of words for step *β.2* is a simple task which is based on the use of word markers (capital letters, underscore character). This is similar to what is done in [10]. Although not perfect, these heuristics give sufficiently good results to be used in a completely automatic way. In cases of problem (lack of word markers in the identifiers), works like [3] could help improve the results.

To compute the lexical difference between two word sequences in step *β.3*, we designed a simple algorithm. The algorithm does not allow more than two contiguous words to be inserted, deleted or changed in a sequences. This reduces greatly the space of search and consequently the time complexity. The algorithm treats the sequences in parallel from left to right, comparing the first word of each sequence at a time. When the first word is different, it first tries to remove this difference with one add(word) or del(word) modification, if it fails, it tries then the chg(word1,word2) modification. The algorithm does not look for a minimal set of modifications and returns the first acceptable set found. It only considers the successful lexical differences containing less than two modifications (consecutive or not). The rational is that with more modifications, the identifiers would be too far apart to be of interest. This heuristic could be fine tuned to take into account the length (in number of words) of both sequences. The fact that the algorithm returns the first lexical difference could theoretically be a problem. However, it would be very difficult to define an hypothetical optimal lexical difference. An other deficiency, on which we will come back (§6), is the absence of a normalized form for the lexical differences.

The next step, *β.4*, computes formal relations between each pair of software components. For structured types, we used the inheritance and composition relations (see also §2.2), for routines, we use the call relation (a routine *calls* another if it has a reference to it in its body, i.e. we used static analysis).

The call and composition relationships are directly represented in the code, whereas inheritance is less direct, it is deduced from the comparison of the structured types' definitions. Thus, it is based on the assumption that the fields' names respect some convention. This is a normal assumption that has been used in various works (e.g. [7, 11]).

Finally, the last step, *β.5*, consists in cross comparing the results from *β.3* and *β.4* and find out which software component pairs appear in both. The pairs are grouped by candidate abstract constructs: All pairs of components with the same lexical difference between them, and the same formal relation between them, will be presented together. We are primarily interested in the candidate abstract constructs with the most identifier pairs.

5 Some Results

We present first the results from our experiment with structured types and then the experiment with routines.

5.1 Structured Types

We found 500 structured types, with 472 different names. For the *inheritance* relationship, there are 278 pairs of related structured types and the algorithm found 22 candidate abstract constructs. For the *composition* relationship, the algorithm found 27 candidate abstract constructs covering 63 pairs from a total of 511 related pairs of structured types.

The first test for this experiment was to establish if we could rediscover the Rec/Part example already discussed in section 3.1. This was actually one of the largest candidate abstract constructs, with 13 pairs for {chg(Rec,Part)}/composition. There are also 14 additional pairs with a longer lexical difference (e.g. {chg(Rec,Part), chg(Widget,Core)}/composition). The test seems conclusive, but we should also mention two apparently opposite abstract constructs: {chg(Core,Widget), chg(Part,Rec)}/composition and {chg(Object,Widget), chg(Part,Rec)}/composition. These come from an "implementation trick" where the component (Part structure) has a pointer to its parent (Rec structure). This feature was mistaken for a composition relation. This is the kind of noise one has to deal with when working with the source code.

Another large candidate is {add(Class)}/composition with 24 pairs. This denotes a pattern of organization where structured types (e.g. _HTStream) have a pointer to a description of their (meta-)class (e.g. _HTStreamClass). Again, the pointer is used in this case to denote something else than composition, nevertheless it helped finding out an interesting feature.

Table 2. Some candidate abstract constructs for structured types and the inheritance relationship.

{chg(Composite,Constraint)}
{chg(Composite,HTML)}
{chg(Composite,Xm),add(Manager)}
{chg(Constraint,HTML)}
{chg(Constraint,Xm),add(Manager)}
{chg(Widget,Composite)}
{chg(Widget,Constraint)}
{chg(Widget,HTML)}
{chg(Widget,Xm),add(Manager)}
{chg(Xm,HTML),del(Manager)}

We singled out some interesting candidates for the inheritance relation in Table 2. They suggest a concept inheritance hierarchy which we represented in Figure 2 (UML notation). The dashed lines correspond to abstract constructs which express only the transitive closure of the actual inheritance relations (plain lines).

The context is especially important here as it is certainly not universally true that HTML (the HyperText Markup Language) is a sub-concept of Widget. One should rather understand that, in this context, HTML denotes an HTML widget which is a kind of constrained composite widget.

One can note that the words Xm and Manager are always associated (in this context): it is either {chg(xxx,Xm),add(Manager)} or {chg(Xm,xxx),del(Manager)}. This is an example of a nominal syntagms, the pair Xm+Manager marks a single concept. This is an advantage of using the opposition heuristic over the repetition heuristic to find the important concepts. The repetition heuristic would need some extra processing (as in [15]) to find such nominal syntagms.

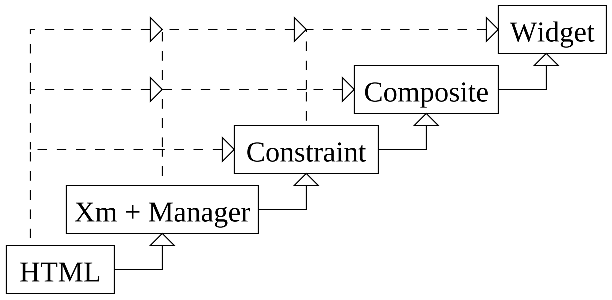


Figure 2. A concept inheritance hierarchy from the Mosaic system and X11 library in the "widget context". Dashed lines show the transitive closure of the inheritance relations.

Having identified an interesting group of structured types, we started to study it more closely. In Table 3, we present the candidate abstract constructs extracted for these concepts with the composition relationship. These constructs are also pictured in Figure 3.

Table 3. Some candidate abstract constructs for structured types and the inheritance relationship.

{chg(Rec,Part)} **- -** *note: "Widget" ∉ context*
{chg(Composite,Core),chg(Rec,Part)}
{chg(Constraint,Core),chg(Rec,Part)}
{chg(Constraint,Composite),chg(Rec,Part)}
{chg(HTML,Core),chg(Rec,Part)}
{chg(HTML,Composite),chg(Rec,Part)}
{chg(HTML,Constraint),chg(Rec,Part)}
{chg(HTML,Xm),add(Manager),chg(Rec,Part)}
{chg(Widget,Core),chg(Rec,Part)}
{chg(Xm,Core),del(Manager),chg(Rec,Part)}
{chg(Xm,Composite),del(Manager),chg(Rec,Part)}
{chg(Xm,Constraint),del(Manager),chg(Rec,Part)}

This example shows an organization schema where each xxxRec structure have a member xxxPart except for WidgetRec which owns a CorePart. The dashed lines shows composition relations that exist but only serve to mimic inheritance in the procedural language (C). We are already familiar with the Rec/Part organization, however, Widget seems to fail to follow the rule. An hypothesis is that Core and Widget are synonymous (or quasi-synonymous) and, maybe by mistake, one is used for the other in this case. This hypothesis is confirmed by two facts:

- Whereas CompositeRec is defined in CompositeP.h, ConstraintRec in ConstraintP.h and XmManagerRec in ManagerP.h, WidgetRec is defined in CoreP.h.

- In CoreP.h we find the following definition:

```
typedef struct _WidgetRec {
    CorePart    core;
 } WidgetRec, CoreRec;
```

One could object that all these structured types form a highly organized piece of code that is not typical of legacy software systems. We will come back to this point in the discussion section (§6).

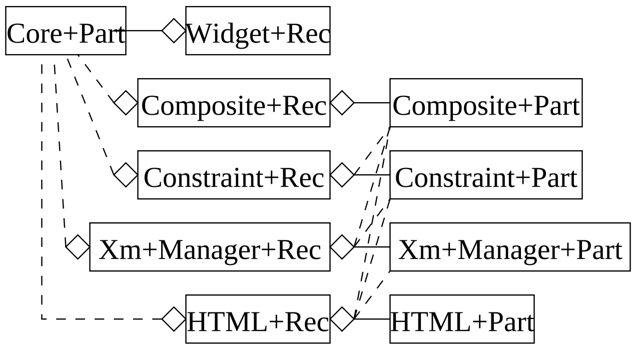


Figure 3. A concept composition graph from the Mosaic system and X11 library. Dashed lines are a consequence of the inheritance relations (see also Figure 2.

5.2 Routines

We also experimented with software components being routines. There are 4331 routines with 3700 different names and 5773 routine pairs caller/called with 1092 different caller routines and 1429 different called routines.

Some interesting candidate abstract constructs are presented in Table 4. The first three show examples of wrapping. The Xmx functions (e.g. XmxAddCallback) are defined in the Mosaic code, the Xt and Xm functions (e.g. XtAddCallback) are part of the X11 library. For example, one of the pair for {chg(Xmx,Xt)}/call comes from the following piece of code:

```
void XmxAddCallback (
        Widget w, String name,
        XtCallbackProc cb, int cb_data)
{
  XtAddCallback ( w, name, cb,
     (XtPointer)_XmxMakeClientData(cb_data));
  return;
}
```

Table 4. Some candidate abstract constructs for routines and the call relationship.

{chg(Xmx,Xt)}
{chg(Xmx,Xm)}
{chg(Xmx,Xm),chg(Make,Create)}
{chg(MoCCI,MCCI)}
{chg(MCCI,MoCCI),del(Request)}

As it is sometimes the case, the interest of this discovery goes far beyond the two examples we discovered (the context of this abstract construct) and may impact one's understanding of all Xmx functions.

The {chg(Xmx,Xm),chg(Make,Create)}/call abstract construct also presents two synonyms: Make and Create, the first being preferred in Mosaic and the second in the X11 library.

The two final candidates are probably an example of an implementation problem. They show calls in both directions from MCCI to MoCCI and the opposite. This suggest a strong coupling between the two concepts. They come from two files cciBindings.c (for MCCI) and cciBindings2.c (for MoCCI) which appear to have the same or very similar functionality. We believe cciBindings2.c is an afterthought addition to cciBindings.c, maybe from another programmer (which would explain the different naming convention).

6 Discussion

An important question about our experiment is its representativity. It could be argued that the examples we gave are particularly lucky ones found in an uncommonly well organized software. We think differently:

- It is true that most of the abstract constructs we found came from the X11 library which seems particularly well structured. However, some abstract constructs did came from Mosaic itself, whether it was by copying the structure of the X11 library (HTML+Rec composed of HTML+Part in section 5.1) or by introducing its own (wrapping example in section 5.2).

- Our confidence in the representativity of this experiment is corroborated by the fact that we found other examples of abstract constructs in a large legacy telecommunication system (see Figure 4).

The kind of conventions our experimental tool is looking for need not be strictly enforced, if an organization scheme appears clearly and has some value, software engineers will naturally tend to follow the lead. This is what appends in the case of Mosaic following an organization scheme from the X11 library.

An other question concerns the interest of the approach for Reverse Engineering. We believe that the uncovering of organizational schemas such as those we presented is important first to help structure a system, but also to keep alive the informal conventions on which these schemas are based. The approach we propose can prove useful in various senses:

- Help discover new abstract constructs and point out to possible things one should look for in the system.
- Help confirm (or infirm) the existence of a potential organizational schema by discovering all its instances. The schema could have been proposed by the method itself, or by a software engineer after informal study of the system.
- Help discover outliers that do not follow the organizational schema (as in the example of Widget+Rec composed of Core+Part, §5.1).

  This is an extremely important point since differing conventions can be more confusing than no convention at all. These outliers could denounce implementation errors, misunderstanding of the abstract construct, or they could signal exceptions, particular configurations that deserve more attention.

The experiment we conducted, uncovered few abstract constructs which could be see as a severe limitation on its utility. We rather believe this is due to the over-simplicity of the two algorithms we used, first to compute the lexical difference, and second to find relation between software components.

- The fact that our algorithm to compute lexical differences does not look for the *optimal* solution does not seem a big problem because the word sequences are usually short. But we see as significant that the algorithm does not *normalize* the lexical differences. Consider the example of two word sequences: {Xm,Manager,Class} and {Composite,Class}. The algorithm can compute one of the two following lexical differences: {chg(Xm,Composite),del(Manager)} or {del(Xm),chg(Manager,Composite)}. But the second lexical difference does not present any clear similarity with other lexical differences found by the tool (e.g. Table 2 in section 5.1) and would cause it to fail to identify a relevant instance of an abstract construct.
- Another important deficiency of our tool is in the search of relationship between the software component. For example one may mimic inheritance in procedural languages by copying the superclass structure in the subclass, which one can adequately detect, or by putting a pointer in the subclass to a superclass part, which would be mistaken for a composition relationship. As already mentioned, pointers between structured types are used to implement many kind of relationship and not only composition. Having a clue on how to differentiate these relationship would benefit not only our research, but the entire Reverse Engineering community.

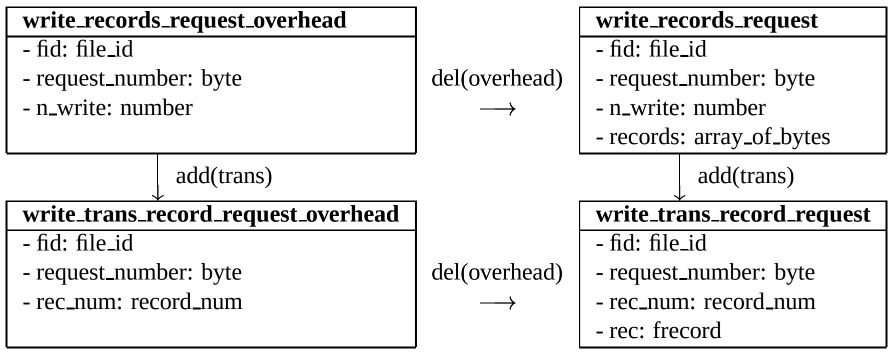
  Many more relationships should be looked for, each one possibly bringing in new information. An example of this is our experiment with the call relationship between routines. Other experiments are needed, with other kind of software components (e.g. variables, files, classes, etc.), or with relationship between two different kinds of software components like variables using types, routines defining or using variables, etc.

Another limitation of this research was that we looked only for abstract constructs containing pairs of software components, but more complex examples exist. We found by chance in a legacy telecommunication system the “square” presented in Figure 4. In this example we have the horizontal lexical difference: {del(overhead)} and the vertical: {add(trans)}. The horizontal relation is inheritance and there is no simple vertical relation (neither inheritance nor composition) between the software components. However the similitude is clearly not fortuitous and such (highly) abstract constructs, if there were to be others, would be of interest to software engineers.

This hints at new research directions where we could drop some of the constraints we used in this research. To automatically find the example above, we need to partly drop the relational part of our abstract constructs (because there is no vertical relation). Going a step further we could completely drop this constraint and look only for sophisticated, purely lexical, constructs. This would discover only concepts without suggesting any relation between them, but would still be of value.

An other important issue is the performance of the algorithm and how it could scale up (Mosaic is at best a medium sized legacy system). We have all reasons to believe that performance will not be an issue. Although the experimental tool was implemented in Prolog (thus interpreted) and not optimized (e.g. it kept recomputing the lexical differences between the software components whereas it could have saved them once for all), execution was fast enough

Figure 4. Example of an elaborated abstract construct on structured types in a telecommunication legacy system.



so that we could work with the tool on a trial-and-error basis, experimenting with different formal relations or relaxing some constraints. This is one of the advantages of working at a more abstract level and therefore have to process less data.

We would like to mention also the interest of the *conceptual* aspect of this research. Things like the small conceptual hierarchy we built in section 5.1 could be useful to understand the domain of the application rather than the application itself. Although it may seem a distant goal at this time, a technique similar to the one we presented could be used to help building an application domain knowledge base.

Finally, the notion of *context* seems important in our approach. We concealed it a bit, but being able to understand this context would be central to a better understanding of the concepts themselves. One should also be able to know in what situations the context can be safely ignored (e.g. when there are many different context for one abstract construct, or as in the case of the sub-system wrapping).

7 Conclusion

In this paper, we proposed a new approach to Reverse Engineering that looks for concepts and relations between them. We introduced the notion of abstract constructs which associate a relation between software components and a lexical difference between their identifiers.

We proposed a simple method to automatically discover abstract constructs. The method was tested on the Mosaic system and the results are very encouraging:

- The tool actually pointed out to informal organization schemas (Rec and Part composition, subsystem wrapping) and helped verify their pertinence.

- The tool also helped in analyzing an outlier of the previously discovered organization schema and we could formulate an hypothesis (Widget act as a synonym of Core in the context studied) that was confirmed by the code.

- The tool helped in discovering a conceptual model of the structured types.

Finally, we discussed some future work to be done in this line of research.

References

[1] N. Anquetil and T. C. Lethbridge. Assessing the Relevance of Identifier Names in a Legacy Software System. In J. H. J. Stephen A. MacKay, editor, *CASCON'98*, pages 213–22. IBM Centre for Advanced Studies, Dec. 1998.

[2] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Working Conference on Reverse Engineering*, pages 235–255. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[3] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:1–21, 1999.

[4] G. Antoniol, G. Canfora, and A. D. lucia. Recovering code to documentation links in oo systems. In *Working Conference on Reverse Engineering*, pages 136–144. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[5] T. K. Arie van Deursen. Identifying object using cluster and concept analysis. In *21^{s}t International Conference on Software Engineering, ICSE'99*, pages 246–55. ACM, ACM press, may 1999.

[6] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. Program Understanding and the Concept Assignement Problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[7] E. Burd, M. Munro, and C. Wezeman. Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity. In *Working Conference on Reverse Engineering*, pages 189–196. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[8] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. Retrieving Information from Data Flow Diagrams. In *Working Conference on Reverse Engineering*, pages 22–29. IEEE, IEEE Comp. Soc. Press, Jul. 1995.

[9] G. Canfora and A. Cimitile. An Improved Algorithm for Identifying Objects in Code. *Software: Practice and Experience*, 26(1):25–48, jan 1996.

[10] B. Caprile and P. Tonella. *Nomen est Omen*: Analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, pages 112–122. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[11] A. Cimitile, A. D. Lucia, G. D. Lucca, and A. Fasolino. Identifying Objects in Legacy Systems. In *5th International Workshop on Program Comprehension, IWPC'97*, pages 138–47. IEEE, IEEE Comp. Soc. Press, 1997.

[12] J. C. S. do Prado Leite and P. M. Cerqueira. Recovering Business Rules from Structured Analysis Specifications. In *Working Conference on Reverse Engineering*, pages 13–21. IEEE, IEEE Comp. Soc. Press, Jul 1995.

[13] J. Johnson. Substring matching for clone detection and change tracking. In *Proc. of International Conference on Software Maintenance*, pages 120–26. IEEE, IEEE Comp. Soc. Press, 1994.

[14] P. Lutsky. Automatic Testing by Reverse Engineering of Software Documentation. In *Working Conference on Reverse Engineering*, pages 8–12. IEEE, IEEE Comp. Soc. Press, Jul 1995.

[15] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[16] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the Automatic Detection of function Clones in a Software System Using Metrics. In *International Conference on Software Maintenance, ICSM'96*, pages 244–53. IEEE, IEEE Comp. Soc. Press, Nov. 1996.

[17] E. Merlo, I. McAdam, and R. D. Mori. Source Code Informal Information Analysis Using Connectionnist Models. In R. Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1339–44. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1993.

[18] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse-engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[19] S. Palthepu, J. E. Greer, and G. I. McCalla. Cliché recognition in legacy software: A scalable, knowledge-based approach. In *Working Conference on Reverse Engineering*, pages 94–103. IEEE, IEEE Comp. Soc. Press, Oct. 1997.

[20] J. Sayyad-Shirabad, T. C. Lethbridge, and S. Lyon. A Little Knowledge Can Go a Long Way Towards Program Understanding. In *5th International Workshop on Program Comprehension*, pages 111–117. IEEE, IEEE Comp. Soc. Press, May 1997.

[21] J. D. Schlesinger and A. A. Reeves. Educating jackal: Cliché library development and use. In *Working Conference on Reverse Engineering*, pages 123–133. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[22] IEEE Technical Council on Software Engineering. `http://www.tcse.org/revengr/`.

[23] T. A. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *Working Conference on Reverse Engineering*, pages 24–32. IEEE, IEEE Comp. Soc. Press, Oct. 1997.