

# Binding Object Models to Source Code: An Approach to Object-Oriented Re-Architecting\*

Johannes Weidl and Harald Gall  
Distributed Systems Group, Technical University of Vienna  
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria, Europe  
{weidl, gall}@infosys.tuwien.ac.at

## Abstract

*Object-oriented re-architecting (OORA) concerns identification of objects in procedural code with the goal to transform a procedural into an object-oriented program. We have developed a method to address the problem of object identification from two different directions: 1) building an object model of the application based on system documentation to ensure the creation of application-semantic classes; and 2) analyzing the source code to identify potential class candidates on the basis of compound data types and data flow analysis. Object model classes are bound to class candidates to prepare a forward biased and thus semantically meaningful program transformation at the source code level. In this paper, we define a similarity measure for classes to enable the binding process. We also describe the constraints and benefits of human intervention in this process. We have applied this method to a real-world embedded software system to identify potential classes; results from the case study are given in the paper.*

## 1. Introduction

Object-oriented re-architecting aims at the transformation of procedural software into functionally equivalent object-oriented software. We have developed a reverse engineering technique called *CORET* (Capsule Oriented Reverse Engineering Technique) that bases upon our previous work [5, 6] but focuses on the object-oriented re-architecting of C to C++ programs.

The essential part within each object-oriented re-architecting is the identification of objects (or statically: classes.) Approaches such as Liu et al. [9, 10] group

data structures with routines that use them as parameters or return values. Yeh et al. [11] combine data structures with global data to form objects with routines as methods. Graph-based approaches such as [1, 4] use reference graphs and subgraphs to form objects on the basis of global variables and functions.

Our object-oriented re-architecting process addresses the object identification from two directions: 1) an object model is built upon documentation of the system under study; and 2) the source code is analyzed and class candidates are identified according to defined criteria (see Section 2.1.) This approach is driven by the object model and, therefore, allows the creation of classes with high application-semantic, i.e. classes that are sound abstractions of the basic application concepts. In this paper, we present the binding process of object models to source code that is one sub-process of our object-oriented re-architecting technique.

*CORET* has been developed using an embedded software system as a real-world case study. For this, persistent data stores as used in our previous approach [5, 6] cannot be used as class candidates. *CORET* works with C and focuses on compound data types (e.g. structs) as object-establishing entities. Because the application-semantic significance of a compound type as a basic building block of the application cannot be assessed at the source code level, a reverse generation of an object model from the source code is not feasible.

For this, we generate an object model from design and implementation related documentation that is ‘close to the source code.’ This means that the object model is intended to exactly represent the procedural system without rigorously introducing high-level object-oriented concepts such as inheritance. This low abstraction level allows a binding of class candidates to object model entities. Following this approach, we minimize the concept assignment problem (see [3].) In the sequel, we will address this object model as ‘ $OM_{design}(OM_D)$ .’

To introduce high-level object-oriented concepts such as inheritance and aggregation we additionally create

---

\*This work, that is pursued in cooperation with Klagenfurt University, is supported by the Austrian Science Fund (FWF), project no. P11.340 ÖMA, and the ESPRIT project ARES, project no. 20477.

an object model based upon the system requirements ( $OM_{requirements}$  or  $OM_R$ ) and map  $OM_D$  to  $OM_R$  classes. This mapping process of OORA is beyond the scope of this paper. To create and maintain different object models we use the object-oriented modeling tool *Rational ROSE*<sup>1</sup> and the Unified Modeling Language (UML) notation.

In the sequel, we describe the main processes in CORET: First, the source code is parsed (*analyzing process*) and the object models ( $OM_D$  and  $OM_R$ ) are created (*modeling process*; see [8].) Then, the binding process is performed. The mapping process matches the  $OM_D$  and  $OM_R$  models to come to an object model that serves as the basis for the system transformation. Finally, in the *transformation process* the procedural source code is transformed by transforming the abstract syntax tree generated in the analyzing process.

The binding process is semi-automated, an engineer is supposed to interact with the binding tool to inject application-specific knowledge and thus improve the binding result. We applied our methodology to a part of a Train Control System which is an embedded software system and present several results later on in this paper.

The remainder of the paper is organized as follows: Section 2 introduces the binding methodology, in Section 3 the similarity measure is described. Section 4 explains the binding process in detail. Section 5 presents our case study, Section 6 draws some conclusions.

## 2. The binding methodology

The binding methodology can be briefly surveyed as follows:

Design and implementation related system documentation is used to create an object model ( $OM_D$ .) Because the modeling process is carried out by an engineer using application-specific information, the application-semantic significance of the objects in the model is trusted.

Class candidates are identified in the source code using defined criteria (see Section 2.1.) The identification step is automated e.g. by using standard reverse engineering tools (we experimented with *Imagix4D*<sup>2</sup>) or by parsing and analyzing the source code with our CORET toolset. For this, the application-semantic significance of class candidates is questionable.

The problem of binding object model entities to source code entities is mapped to the problem of finding matches *class* to *class candidate*. This problem in turn is addressed by defining a similarity measure between classes. For this, we understand class candidates as classes (i.e. variables as attributes, procedures and functions as methods) and base a

$c$	class name
$an$	attribute names
$at$	attribute types
$or$	operation return types
$on$	operations names
$opn$	operation parameter names
$opt$	operation parameter types
$asn$	association names
$ast$	association types
$\#a$	number of attributes of the class
$\#o$	number of operations of the class
$\#as$	number of associations of the class

**Table 1. Class signature parts**

binding between classes and class candidates on their *class similarity*.

### 2.1. Class signatures and class candidates

We introduce the notion of a *class signature*.

**Class signature.** The *class signature*  $s_c$  of a class  $c$  consists of *class signature parts* shown in Table 1. We use class signatures as the basis for determining *similarity* between classes. We define the following operations on the set  $S$  of class signatures:

- $s_{c_i} \subseteq s_{c_j} \Leftrightarrow$  (all class signature parts are equal (i.e. same names and types of attributes, operations, parameters, and associations) except  $\#a$ ,  $\#o$ , and  $\#as$  for which the following condition holds:  
 $\#a_{c_i} \leq \#a_{c_j} \wedge \#o_{c_i} \leq \#o_{c_j} \wedge \#as_{c_i} \leq \#as_{c_j}$ )
- $s_{c_i} = s_{c_j} \Leftrightarrow (s_{c_i} \subseteq s_{c_j} \wedge s_{c_j} \subseteq s_{c_i})$
- $s_{c_i} \subset s_{c_j} \Leftrightarrow (s_{c_i} \subseteq s_{c_j} \wedge s_{c_i} \neq s_{c_j})$

Furthermore, we define the notion of a *class candidate*.

**Class candidate.** An *class candidate* is a set of source code elements that is understood as a class but which application-semantic significance is not trusted. Class candidates are automatically identified in the source code and are an intermediate representation of source code elements. We consider the following class candidates:

- Compound data types together with related procedures and functions (e.g. structs together with procedures and functions that use and/or manipulate the data type.)
- Sets of global variables grouped by data flow analysis together with the corresponding procedures and functions. (e.g. global variables, that are jointly used and/or manipulated in procedures and functions.)

<sup>1</sup>*Rational ROSE* is a trademark of Rational Software Corp.

<sup>2</sup>*Imagix4D* is a trademark of Imagix Corp.

### 3. A similarity measure for classes

The binding process is supposed to assign  $OM_D$  classes to class candidates. Such an assignment should happen, if the application-semantics of the two entities is equal or, at least, similar. For this purpose, we define a similarity measure between classes based on a class signature comparison. We define a similarity measure function  $f_{SM} : (s_{c_i}, s_{c_j}) \mapsto [0, 1]$ .  $f_{SM}$  assigns each pair of class signatures a real value between 0 and 1 expressing their similarity. From a high similarity in the class signatures (i.e. same number, names, types of attributes, same method signatures, etc.) we conclude a high (application-) semantic similarity of the entities involved. We claim that entities with a similar semantic content in many cases have similar class signatures and vice versa. We use the following class signature parts for measuring the similarity of classes:  $c$ ,  $an$ ,  $at$ ,  $or$ ,  $on$ ,  $opn$ ,  $opt$ ,  $\#a$ ,  $\#o$  (see Table 1.) We now introduce the similarity measure step by step:

$$\begin{aligned} sf_{sum} &= sf_c * w_c + \\ &+ sf_{an} * w_{an} + sf_{at} * w_{at} + \\ &+ sf_{or} * w_{or} + sf_{on} * w_{on} + \\ &+ sf_{opn} * w_{opn} + sf_{opt} * w_{opt} \end{aligned}$$

First, we introduce the terminology by giving some definitions:

**Similarity factor  $sf$ .** The similarity between class signature parts expressed as a real value between 0 and 1 is called *similarity factor  $sf$* . For each class signature part  $x$ , a similarity factor  $sf_x$  is computed.

**Similarity factor weight  $w_x$ .** The *similarity factor weight  $w_x$*  as a real value between 0 and 1 is used to weigh each similarity factor in the similarity measure computation.

**Weight vector  $w$ .** The *weight vector  $w$*  consists of all similarity factor weights  $w_{x_i}$ . The sum of all weight vector components is supposed to be 1.

$$w = (w_c, w_{an}, w_{at}, w_{or}, w_{on}, w_{opn}, w_{opt})$$

The similarity factors between the class signature parts are computed and summed up in  $sf_{sum}$  considering a weight factor for each  $sf$ . This means, that the influence of the similarity factors ( $sfs$ ) on the sum can be controlled by using a weight vector  $w$ . Using weights, the similarity measure result can be focused e.g. to attributes by setting the  $sf$  weights accordingly.

Next, we introduce  $min_a$  and  $max_a$  as the minimum and maximum number of attributes regarding two classes  $c_1$  and  $c_2$ . The function  $\#a$  returns the number of attributes of a class. For operations,  $min_o$ ,  $max_o$ , and  $\#o$  are defined accordingly.

$$\begin{aligned} min_a &= \min(\#a(c_1), \#a(c_2)) \\ max_a &= \max(\#a(c_1), \#a(c_2)) \end{aligned}$$

We further introduce  $sf_{max}$ , which is the maximum  $sf$  possible if the two classes compared are of different granularity (i.e. the number of attributes and operations differs.)  $sf_{id}$  is the maximum  $sf$  possible if the two classes have the same granularity (for this, it is called ‘ideal  $sf$ ’.)

$$\begin{aligned} sf_{max} &= w_c + min_a * (w_{an} + w_{at}) + \\ &+ min_o * (w_{or} + w_{ot} + w_{opn} + w_{opt}) \\ sf_{id} &= w_c + max_a * (w_{an} + w_{at}) + \\ &+ max_o * (w_{or} + w_{ot} + w_{opn} + w_{opt}) \end{aligned}$$

We distinguish between the similarity of classes 1) considering granularity differences ( $osf$ ) and 2) not considering granularity ( $osf_{po}$ .)

$$osf = \frac{sf_{sum}}{sf_{id}} \quad osf_{po} = \frac{sf_{sum}}{sf_{max}}$$

**Part-of overall similarity factor  $osf_{po}$ .** The similarity between two classes without considering the difference in granularity is called *part-of overall similarity factor  $osf_{po}$* .  $osf_{po} = 1 \Leftrightarrow (s_{c_i} \subseteq s_{c_j} \vee s_{c_j} \subseteq s_{c_i})$  for two class signatures  $s_{c_i}$  and  $s_{c_j}$

**Overall similarity factor  $osf$ .** The similarity between two classes considering the difference in granularity is called *overall similarity factor  $osf$* .

$$osf = 1 \Leftrightarrow s_{c_i} = s_{c_j} \text{ for two class signatures } s_{c_i} \text{ and } s_{c_j}$$

In the sequel, we describe the different similarity check techniques for the class signature parts.

#### 3.1. Syntactic similarity

For measuring the similarity of class names, attribute names, operation names, and parameter names *fuzzy text comparison* is used [2]. When comparing strings, fuzzy text comparison does not give a crisp result but divides the string into substrings of a fixed length<sup>3</sup>, counts the substring matches, and accordingly computes a real number between 0 and 1 that expresses the *orthographic similarity* of the strings. The similarity  $sf_c$  between the name  $c_c$  of class  $c$  and the name  $c_{cc}$  of class candidate  $cc$  is defined as follows:

$$\begin{aligned} sf_c &= \text{fuzzytextcompare}(c_c, c_{cc}) \\ sf_{an_{ij}} &= \text{fuzzytextcompare}(an_i, an_j) \end{aligned}$$

$sf_{an_{ij}}$  is computed for each pair  $(an_i, an_j)$ , where  $an_i$  is the name of the  $i$ -th attribute of a class  $c$  and  $an_j$  is the name of the  $j$ -th attribute of a class candidate  $cc$ . The  $min_a$  highest  $sf_{an_{ij}}$  values are summed up in  $sf_{an}$ . An attribute name is allowed to contribute only once to the sum, that

<sup>3</sup>We use so-called tri-grams (substrings of length 3.)

$tm_f$ :

	char	int	float	...
char	1			
int	.6	1		
float	0	.2	1	
...				1

$tm_d$ :

	design	scalar	sized
implem.			
scalar		1	1/size
sized		1/size	size</size>
pointer		.9	.9

**Figure 1. Fundamental type matrix  $tm_f$  and derived type matrix  $tm_d$**

means if  $sf_{an_{ij}}$  is used in the sum  $sf_{an}$ , all  $sf_{an_{ik}}$  with  $k \neq j$  and  $sf_{an_{lj}}$  with  $l \neq i$  must not be considered in the further  $sf_{an}$  computation. Thus, only the best match of a class attribute name to a class candidate attribute name is considered in the  $sf_{an}$  sum. For operation and parameter names, the  $sf$  computation is performed analogously.

### 3.2. Type similarity

The similarity of types is based on a semantic comparison using *type similarity matrices*. We use two matrices, the *fundamental type similarity matrix* and the *derived type similarity matrix*. For each two fundamental (or built-in) types (such as `int`, `char`, etc. in C), the fundamental type similarity matrix specifies their similarity as a real value between 0 and 1. We established such a matrix by heuristically assessing the semantic similarity of types. As an example, for the pair (`int`, `float`) we use 0.2.

The *derived type similarity matrix* is used when comparing a derived type (i.e. *sized*<sup>4</sup> and *pointer*) with a *scalar*<sup>5</sup> type or when comparing two derived types. The  $OM_D$  types are either scalar or sized. In the source code, the derived type class *pointer* additionally occurs. When an entity is of a user-defined type, the type has to be examined for its components recursively. Figure 1 shows the fundamental and derived type matrices.

$$\begin{aligned} sft_{fundamental} &= tm_f[ftype(a_i), ftype(a_j)] \\ sft_{derived} &= tm_d[dtype(a_i), dtype(a_j)] \\ sft_{at_{ij}} &= sft_{fundamental} * sft_{derived} \end{aligned}$$

$sft_{fundamental}$  and  $sft_{derived}$  are computed for each pair  $(a_i, a_j)$ , where  $a_i$  is the  $i$ -th attribute of a class  $c$  and

<sup>4</sup>A sized type is a container for a well-defined number of data of a scalar type, e.g. `int b[6]`;

<sup>5</sup>e.g. `int a;`

$a_j$  is the  $j$ -th attribute of a class candidate  $cc$ .  $ftype(e)$  is a function returning the fundamental type of an entity  $e$ .  $dtype(e)$  returns the derived type of  $e$ .  $sft_{fundamental}$  and  $sft_{derived}$  are multiplied. Thus, since the factors are  $\leq 1$ , the similarity between two typed entities is reduced according to the factors. The specific  $sf_{at}$  computation is analogous to the  $sf_{an}$  computation (see Section 3.1.) For operation return types and parameter types, the  $sf$  computation is performed as described above.

Using type matrices, the similarity measure is customizable to different applications and domains by adopting the matrix entries accordingly. By defining the class signature part  $sfs$  in this way, we can be sure that a higher similarity leads to a higher  $osf$  and that  $f_{SM}(s_{c_i}, s_{c_i}) = 1$ .

## 4. The binding process

The result of the similarity check is the *osf table*. The *osf* table contains *osf* and *osf<sub>po</sub>* for each pair of class and class candidate. Additionally, it stores the weight vector used for the computation and the single signature part  $sfs$ .

The *osf table* is the starting point for the binding process. In the binding process, certain system functionality ( $OM_D$  classes) is linked to its implementation (parts of the source code.) Human interaction is used in the binding process to resolve binding conflicts and optimize the binding result by introducing application and domain knowledge. A function assessing the quality of a binding result quantitatively could have a local maximum when all  $OM_D$  classes are bound to class candidates. A global maximum would be a local maximum, where all bindings have the maximum *osf* possible.

The result of the binding process is the *binding table*. The binding table stores information, which classes correspond to what class candidates. This means, it is a set of explicit links between  $OM_D$  model entities and source code entities that is used to transform procedural into object-oriented source code. Furthermore, the binding table represents high-level documentation which can be used in system maintenance. Using the binding table, it is possible to directly navigate from model entities to the position in the source code where they are implemented.

### 4.1. Binding process strategies

Different strategies can be used to come to a binding result. Consider the following simple binding strategy:

1. Sort the *osf* table in descending order according to the *osf*-value.
2. Start with the first element in the *osf* table, i.e. the element with the highest *osf*.
3. Move the *osf* table entry to the binding table.
4. Mark all *osf* table entries containing the object or object candidate of the binding just made.

- If an osf table entry remains unmarked continue with step 3.

This strategy has a major drawback in that no human intervention is possible to resolve binding conflicts and prevent incorrect (qualitatively in the sense of unreasonable) bindings. For this, we extended the above binding strategy (for more details on binding strategies see [7].) An engineer is allowed to define *urged bindings* before the beginning of the process. Urged bindings are bindings, the engineer knows about a priori because of his/her application or domain specific knowledge. Furthermore, bindings can be defined that must not be made and are dropped when occurring in the binding process (*forbidden bindings*.)

The binding algorithm presented above is done iteratively. After each step the engineer decides which bindings will be *fixed*. Fixed bindings are added to the list of urged bindings for the next iteration step. If there is a binding conflict (e.g. there are two source code entities that have about the same *osf* with the same model entity), the engineer is asked to resolve the conflict and fix the correct binding. The process stops when all bindings are fixed. This is the binding strategy used in our case study.

## 5. Case study

Our case study is part of a Train Control System that is a real-world embedded software system provided by an industrial partner. The system under study is one version of a family of systems which are safety-critical and have strong timing considerations. The software is programmed in C and Assembler and has to run on different development and target environments. The system controls high speed train movement and realizes precision stops in metros. We applied the binding methodology to a part of the system dealing with the reception, decoding, and distribution of messages. For this, we have implemented a binding tool in JAVA that automates the similarity check and supports the binding process. We present a small example from the case study.

The following alphabetically sorted binding table shows the correct (qualitatively in terms of most reasonable) binding of a subset of  $OM_D$  classes to a subset of class candidates<sup>6</sup>:

```
class candidate : class
CDDATA : DCD_Message
DMDATA : DDM_Message
FB1_VIRTUAL : TR_Message
FB2_TYPE : ML_Message
LCDATA : DLC_Message
SPDATA : DSP_Message
```

<sup>6</sup>For confidentiality reasons, we have changed the class and class candidate names. The values shown in the examples were computed using the original names.

In the sequel, we use the weight vector  $w = (.1, .15, .15, .15, .15, .15, .15)$ , which gives all class signature parts about the same weight. By modifying the weights the measure can be biased to particular class signature parts. The similarity check results in an osf table:

```
class candidate:class  sfc  sfan  sfat      osf
                        sfor  sfon  sfopn  sfopt  osfpo
...
SPDATA:DSP_Message  0.091  0.83   1      0.36
                        1  0.12   0      0  0.7
SPDATA:DLC_Message  0  0.56  0.78   0.28
                        1  0.12   0      0  0.55
...
```

Using the binding strategy of Section 4.1, the osf table given above leads to the following binding table:

```
class candidate : class      osf  osfpo
LCDATA : DLC_Message      0.5  0.79
DMDATA : DDM_Message      0.39 0.64
FB2_TYPE : ML_Message     0.38 0.66
CDDATA : DSP_Message      0.38 0.69
FB1_VIRTUAL : TR_Message  0.3  0.62
SPDATA : DC               0.26 0.39
```

The suggested binding of CDDATA to DSP\_Message with *osf* 0.38 is incorrect because the correct binding partner (class DCD\_Message) has only two attributes (because of a lack of information in the design documentation.) This granularity difference causes a small *osf* between CDDATA and DCD\_Message.

The *osf*-value between SPDATA and DSP\_Message (which is the correct binding) is only 0.36, again because of a granularity difference in the number of attributes of these entities. Thus, CDDATA and SPDATA are bound incorrectly. This suboptimal binding will be improved in the sequel.

After the first iteration, the engineer fixes the binding LCDATA to DLC\_Message by adding it to the list of urged bindings<sup>7</sup>. Since the next three bindings have about the same *osf*, the engineer uses his/her application knowledge, system documentation, or the source code to reason about the correctness of the bindings. We assume, the engineer finds out, that the binding CDDATA to DSP\_Message is incorrect. He forbids the binding and starts the next step of the binding process:

```
class candidate : class      osf  osfpo
LCDATA : DLC_Message      1    1
DMDATA : DDM_Message      0.39 0.64
FB2_TYPE : ML_Message     0.38 0.66
SPDATA : DSP_Message      0.36 0.7
FB1_VIRTUAL : TR_Message  0.3  0.62
CDDATA : DC               0.27 0.39
```

<sup>7</sup>A pair of entities in an urged binding is assigned 1 for its *osf* and *osf<sub>po</sub>* values.

By introducing a ‘fixed point’ (certainty) into the process, the binding table changes. Because of the granularity difference of CDDATA and DCD\_Message mentioned above, CDDATA is bound incorrectly again, in this case to the class DC. This has to be resolved via human interaction.

Instead of forbidding the binding CDDATA to DSP\_Message, the engineer could as well have found out, that the binding CDDATA to DCD\_Message is the correct binding and could have decided to add it to the list of urged bindings:

class candidate : class	osf	osfpo
CDDATA : DCD_Message	1	1
LCDDATA : DLC_Message	1	1
DMDATA : DDM_Message	0.39	0.64
FB2_TYPE : ML_Message	0.38	0.66
SPDATA : DSP_Message	0.36	0.7
FBI_VIRTUAL : TR_Message	0.3	0.62

This directly leads to the correct binding result. After approving all remaining bindings as fixed, the process stops and the binding is finished.

## 5.1. Results

On the basis of our examinations of the case study, we get the following results:

- The binding results based on our similarity measure exceeded our expectations since it seems indeed reasonable to conclude an application semantic similarity of classes and class candidates from a syntactic/semantic similarity of class signatures.
- Fixing bindings by an engineer leads to convergence in the binding process, i.e. the process stops when all bindings have been fixed. By suggesting bindings, incorrect suggestions can be proved and resolved. By fixing a binding, certainty about the binding is introduced and the further process only concentrates on the binding candidates that are left.
- Meaningful entity names are important but not a precondition in our similarity measure.
- The attribute comparison without considering operations leads to good results.
- Because our case study is an embedded software system, the operation comparison does not have big impact. We expect information systems to have a higher granularity in operations than embedded systems, yielding a better result in the operation similarity check.

## 6. Conclusion

In this paper, we presented a method to address the problem of object identification in re-architecting C to C++ programs: object model entities are bound to class candidates

derived from the source code to prepare a forward-driven and thus semantically meaningful program transformation. We defined a similarity measure for classes based on class signatures and explained how this measure can be used to set up a binding process.

Binding, as introduced, is a semi-automated method for forward-driven object identification. Human interaction is necessary to resolve binding conflicts and to detect and resolve incorrect bindings.

Our examinations of a real-world embedded software system have shown that our binding strategies lead to good results, i.e. the identification of application-semantic classes. The integration of human interaction in the binding process gives the binding result high reliability.

## 7. Acknowledgments

We are grateful to Roland Mittermeir, René Klösch, and Dominik Rauner-Reithmayer for many interesting discussions and valuable comments.

## References

- [1] B. L. Achee and D. L. Carver. A greedy approach to object identification in imperative code. *2<sup>th</sup> International Workshop on Program Comprehension (IWPC '94)*, Washington, D.C., USA, pages 4–11. Institute of Electrical and Electronics Engineers, November 1994.
- [2] R. C. Angell, G. E. Freund, and P. Willett. Automatic spelling correction using a trigram similarity measure. *Information Processing & Management*, 19(4):255–61, 1983.
- [3] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.
- [4] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software Practice and Experience*, 26(1):25–48, January 1996.
- [5] H. Gall, R. Klösch, and R. Mittermeir. Object-oriented re-architecting. *5<sup>th</sup> European Software Engineering Conference (ESEC '95)*. Springer Verlag, Berlin, September 1995.
- [6] H. Gall, R. Klösch, and R. Mittermeir. Using domain knowledge to improve reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 6(3):477–505, 1996.
- [7] H. Gall and J. Weidl. Binding object models to source code: an approach to object-oriented re-architecting. Technical report, Distributed Systems Group, Technical University of Vienna, September 1997.
- [8] H. Gall and J. Weidl. Object-model driven abstraction-to-code mapping. Technical report, Distributed Systems Group, Technical University of Vienna, December 1997.
- [9] S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. *IEEE Conference on Software Maintenance*, pages 266–71, November 1990.

- [10] R. M. Ogando, S. S. Yau, S. S. Liu, and N. Wilde. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance: Research and Practice*, 6:261–83, 1994.
- [11] A. Yeh, D. Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. *2<sup>nd</sup> Working Conference on Reverse Engineering (WRCE '95)*, pages 227–36. IEEE Computer Society Press, July 1995.