Object-Model Driven Abstraction-to-Code Mapping

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

Abstract

In object-oriented re-architecting, we face the problem of improving the maintainability of procedural source code to facilitate software evolution. We do this by transforming the procedural code into an object-structure employing encapsulation to the legacy data structures and their related procedures. To handle the concept assignment problem, we established a stepwise abstraction-to-code mapping via different object models based on criteria such as the kind of information used in the modeling process, model granularity, and model abstraction. This mapping is object-model (and thus forward-) driven rather than source-code (or reverse-) driven and therefore enables the specific use of application and domain knowledge. For that, an object-model driven approach promises a high-level semantic class structure for an improved software evolution process. We have applied our approach to a real-world embedded software system to identify potential objects; several results from the case study are given in the paper.

1 Introduction

We denote the transformation of procedural software into functionally equivalent object-oriented software as *Object-Oriented Re-Architechting* (OORA). Our OORA approach *CORET* (Capsule Oriented Reverse Engineering Technique) is based upon our previous work [4, 6] but focuses on the transformation of C to C++ programs.
Improving the maintainability of a system can extend its life-time considerably and decrease maintenance costs; both are objectives essential for the success of real-world applications. Furthermore, our OORA approach aims at generating objects that locate and encapsulate logically interrelated code

to improve cohesion and minimize coupling. This facilitates a potential reuse process.

We clearly state that our approach has limitations especially concerning the reverse engineering of (sophisticated) inheritance hierarchies together with polymorphism and strict encapsulation concerning the generation of private class attributes. However, experience shows that industry is reluctant in re-implementing existing software systems. We have developed a functionality-preserving semi-automated transformation process as an alternative.

In re-engineering an existing system written in C, we face the *concept assignment problem* [1, 2], i.e. the problem of relating source code entities to real-world and application-domain concepts[1]. In OORA, we additionally have to group source code entities to classes representing sound abstractions in the context of object-orientation. This task is denoted as *object identification*. In literature, several approaches to this task can be found, e.g. Liu et al. [8, 9] or Yeh et al. [12], based on different criteria for the identification.

In our CORET process we use a model-driven rather than a source code-driven approach to the problem of object identification. We use different object models (OMs) to represent the application at different levels of abstraction[2]. Different sources of information are used for the modeling process, such as design documentation, requirements, and information from application experts. The object models are then related to the source code in a mapping step ('abstraction-to-code mapping') and serve as the basis for the transformation. Furthermore, the models represent additional system documentation and can be used in a re-implementation process. Objective of this paper is to show how object models can be utilized in the OORA process and how they are related to resolve the abstraction-to-code mapping problem.

In CORET, we use different tools: 1) For the modeling, we use Rational's *ROSE*[3]; 2) To reverse engineer C code, we use *Imagix4D*[4] as well as our own parser; 3) The abstraction-to-code mapping process is as well tool-supported. We will briefly survey these tools in Section 4.

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 two languages (C, Assembler) and has to run on different development and target environments. The system controls high speed train movement and realizes precision stops in metros. The code size is approximately 150K LOC (Lines of Code), with the software implemented as state automatons as described in the SDL specifications of the system. Time critical parts are implemented in Assembler, while the rest of the software is implemented in C. The source code is well commented and the software documentation of the studied parts of the system were available. We will present some examples concerning this software system.

[1] We denote the representation of such concepts as *abstractions*.

[2] Currently, we use OMT as modeling notation [11].

[3] *Rational ROSE* is a trademark of Rational Software Corp.

[4] *Imagix4D* is a trademark of Imagix Corp.

The paper is organized as follows: Section 2 introduces the different object models, Section 3 describes the abstraction-to-code mapping process in detail. In Section 4, we explain the tools we use. Section 5 gives an outlook to further work and Section 6 draws some conclusions.

2 The use of object models in CORET

Figure 1 shows a sketch of the CORET tool with its main processes in terms of a data flow diagram. First, the source code is parsed ("Analyzing") and different object models are created ("Modeling"). Then, the binding process is performed, i.e. the source code entities are associated with their corresponding classes in a dedicated object model (see Section 2.1). The mapping process matches the object models to obtain a *target* object model (see Section 2.4) that serves as the basis for the system transformation. Finally, the procedural source code is transformed into object-oriented code. This paper concentrates on the modeling process with all its object models involved. The detailed binding (see [5]), mapping, and transforming (see [4, 6]) are beyond the scope of the paper.

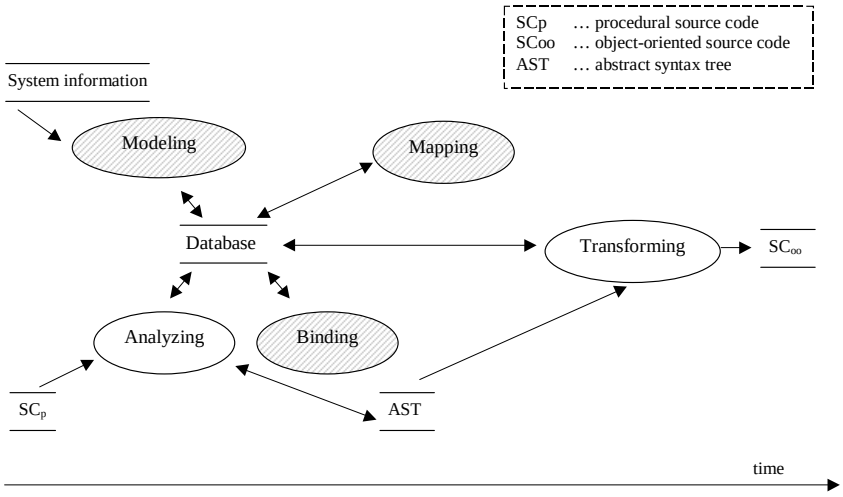

Figure 1: CORET tool sketch

Application modeling as a goal driven activity establishes no one-to-one correspondence between a given application and 'its' application model. For this, CORET utilizes a set of application models.
The object models used in CORET are grouped by their level of abstraction compared to the source code (see Figure 2):

1. The binding process is done between the procedural source code and OM_{design} (Section 2.1).
2. The mapping process is done between OM_{design} and $OM_{requirements}$ (Section 2.2).

3. $OM_{requirements}$ can be used as the basis for creating $OM_{architecture}$, which is intended to represent the system's 'architecture' in terms of high-level concepts (Section 2.3).
4. OM_{target} is the result of the mapping process between OM_{design} and $OM_{requirements}$ (Section 2.4).

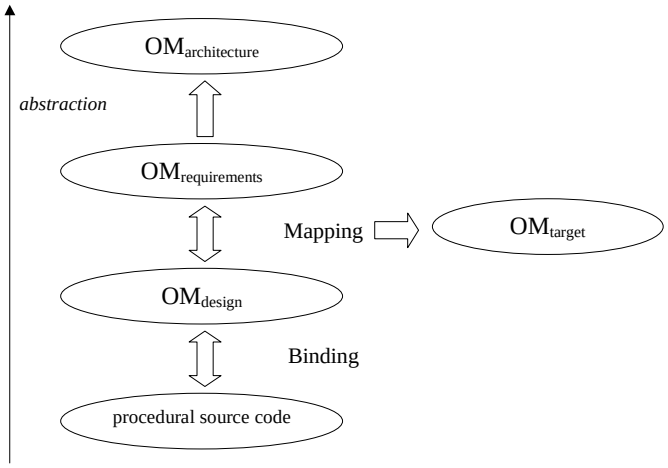

Figure 2: The object models in CORET

The models differ in various aspects: the source of information used for the modeling; thus, the quality of information used in the modeling process; the degree of abstraction and the model granularity[5]; the incorporation of object-oriented concepts (especially inheritance); the relationship to the legacy system to be re-architected and to the target system transformed; the annotations of the model entities (model entities are basically attributes, methods, classes, and associations), the task of the object model in the process; the mode of the model creation (e.g. manually). Each OM can exist in various versions due to differences in e.g. reliability of modeling information, granularity, or completeness. Each individual model is indexed and denoted as OM_i.
We predominantly use the OMT object model [11] and interaction diagrams of the dynamic model, we currently neglect the functional model (i.e. object-interaction diagrams, data-flow diagrams). The functional model has only little influence in the task of abstraction-to-code mapping since this part of the OORA process concentrates on the static structure of the system rather than on dynamical aspects such as the exchange and flow of data. We do utilize use-cases for refining the models. In future, we will switch to UML [3]. Section 2.1 to Section 2.4 describe the CORET object models in detail.

2.1 OM_{design}

OM_{design} is based on application-specific knowledge introduced by design and implementation related documentation and information given by application

[5]High granularity means a high degree of detail in the model and vice versa.

experts. For this, its abstraction level is low compared to the other models depicted in Figure 2. Low abstraction means that the model is 'close' to the source code in terms of containing design and implementation related concepts (e.g. 'list' as an implementation abstraction of a data container). Furthermore, OM_{design} is not modeled strictly applying all the object-oriented mechanisms available. Since inheritance is not explicitly stated in non-oo documentation and code, OM_{design} usually does not contain inheritance hierarchies.

The careful and restrictive use of oo modeling mechanisms—especially inheritance—leads to an object model that is structurally highly related to the source code. OM_{design} is an object-oriented view of the procedural system under study and exhibits the *virtual class structure* of the system. This virtual class structure originates from understanding the real-world, design, and implementation concepts of the procedural system as interrelated classes following the object-oriented paradigm.

By modeling OM_{design} in a forward-driven step the application-semantic content of the classes in the model can be trusted. This means, that those classes are sound abstractions of the basic functionality of the system because of the introduction of application and domain-specific knowledge by a human engineer.

In a reverse engineering step, so-called *class candidates* are automatically recovered from the procedural source code. As class candidates we consider compound data types together with related procedures and functions as well as sets of global variables grouped by data flow analysis[6]. Because of the automated reverse generation, the application-semantic content of class candidates is questionable meaning that a considerable part of the abstractions recovered will be inferior. By binding object candidates to forward-modeled classes we yield a semantically meaningful class structure of the procedural system. This is the essence of our object identification approach and distinguishes it from other—especially fully-automated—approaches.

From the above discussion it should be clear that the structural similarity between OM_{design} and the procedural source code is essential when it comes to binding. Obviously, a higher structural similarity leads to a better binding result, i.e. a higher number of class candidates bound to classes.

According to the type of information used in the modeling process we identify three categories of OM_{design} classes according to their existence in the procedural source code (see Figure 3).

1. **Existing classes.** Existing classes (E-classes) are based on a concept that can be found in the source code in an appropriate *procedural instantiation*. This means that parts of the procedural source code implement the concept. Existing classes are modeled as reliable or correct information from the documentation or from an application expert.

2. **Possibly existing classes.** Possibly existing classes (P-classes) are model classes that might have no instance in the source code. Possibly

[6]For a more detailed definition of the term 'class candidate' see [5] (therein called 'object candidate').

existing classes originate from unreliable information.

3. **Non-existing classes.** Non-existing classes (N-classes) have no correspondence in the source code. Such classes are used to yield a complete and comprehensive view of the system. For example, if a specific functionality (e.g. encoding/decoding) is done in hardware, there is no or only few code dealing with the concept of an 'En/Decoder'. Class B in Figure 3 depicts such a class: A part of the concept is implemented in software, the other one is implemented in hardware or even missing in the current version of the system.

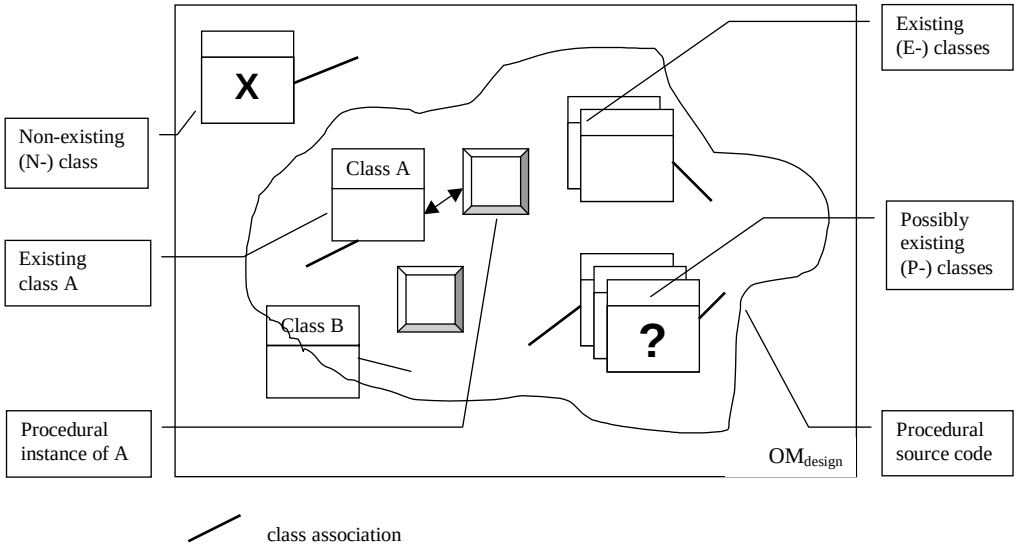

Figure 3: Class categories in OM_{design}

The more E- and N-classes and the less P-classes are identified in the model, the better the binding result will be. The type of information (reliable, unreliable) often cannot be assessed at the time of modeling—if ever. Thus, the modeling engineer often cannot be sure which class category a class belongs to. However, the knowledge of E- and N- classes builds up fixed points in a modeling process that is characterized by inherent uncertainty. These fixed points can be used to resolve subsequent uncertainties in later steps of the OORA process.

The category of a class (E, P, N) is stored in the *class category annotation*. In general, all model entities, i.e. basically classes, attributes, methods, parameters, and associations can be annotated. Model entities can be annotated with a detailed description of the information source that led to the creation of the model entity.

OM_{design} is an annotated object model of low abstraction containing explicit links to the procedural source code as the result of our model-driven abstraction-to-code mapping process. The explicit relation (object model – procedural source code) enables a wide range of possibilities to restructure

the system using oo modeling mechanisms on the basis of a comprehensible object model.

2.2 $OM_{requirements}$

On the one hand, OM_{design} is generated from design and implementation related system documentation and is intended to be modeled at a low abstraction level to obtain a better binding result. On the other hand, $OM_{requirements}$ is generated from the system requirements and domain knowledge and concentrates on all aspects of object-orientation. Since OM_{design} does not consider high-level object-oriented concepts, $OM_{requirements}$ can be used to optimize the OORA process by incorporating fully object-oriented structures (especially inheritance) thus increasing the quality of the *target* system (i.e. the system after the re-architecting).
It is important to model $OM_{requirements}$ according to the specific legacy system requirements to get an accurate model of the system under study. It is not valid to incorporate adapted requirements: Especially information of human experts tends to be biased not to the original requirements of the legacy system but to requirements the system is supposed to meet at the time of inspection.
The granularity of $OM_{requirements}$ in terms of the absolute number of abstractions or concepts in the model is lower compared to OM_{design} because $OM_{requirements}$ does not contain design and implementation related detail. The granularity in terms of a specific abstraction will be higher because of the use of, for instance, inheritance hierarchies or aggregation constructs.
In the mapping process of CORET, a target object is generated by raising the OM_{design} abstraction level using $OM_{requirements}$ as 'optimal' template in terms of object-orientation. The mapping process involves the folding of hierarchies and constructs and similarity checks between model classes analogous to the binding process [5].
To clarify the $OM_{requirements}$ – OM_{design} difference, we present an example from our case study. We focus on two particular subsystems called 'Receiver' and 'Decoder'. The Receiver subsystem collects data from the environment, generates a encoded message and sends the message to the the Decoder, which decodes the message and sends the data on to other subsystems. The Receiver subsystem communicates with the Decoder subsystem using a Data Channel, over which the data is sent as messages. The communication is triggered by a Control Channel connecting Receiver and Decoder. The concept of channels is quite hardware related focusing on the medium rather than on the actual data sent across. While a channel abstraction is used in OM_{design}, $OM_{requirements}$ abstracts the hardware related concepts to the more data centered abstraction of PDUs (Protocol Data Units). This shift in abstractions is shown in Figure 4, which is a meta-diagram showing the difference in abstractions rather than being part of an OMT model.

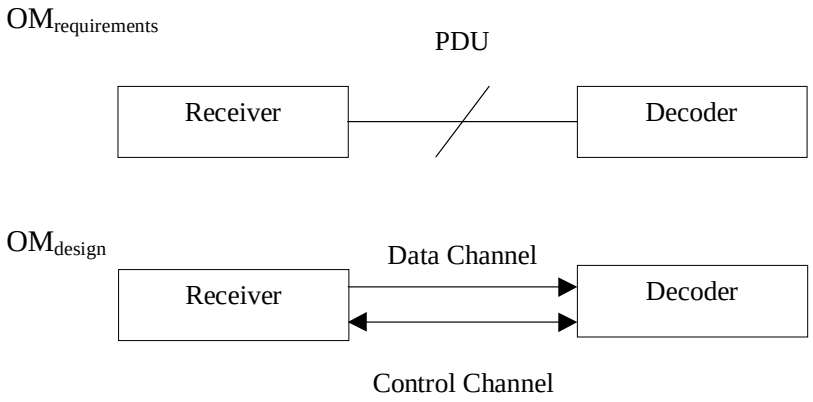

Figure 4: Abstraction differences in $OM_{requirements}$ and OM_{design}

2.3 $OM_{architecture}$

During the OORA process we experienced that $OM_{requirements}$ can be further abstracted to an architectural description of the system. It turned out to be an appropriate instrument in discussing about the high-level design of the system or in creating a reference architecture for a whole family of systems. For example, in $OM_{architecture}$ functional and non-functional system requirements can be identified and modeled individually.
$OM_{architecture}$ is not directly involved in the CORET process, it rather represents documentation which can be used to reason about the system from a high-level view. It can be the basis of restructuring the system to meet adapted requirements in a reengineering activity.
In our case study we further abstracted the PDU concept to the concept of 'Intermodule Communication', which is an encapsulated component of the system and can be designed and implemented independently in various ways. In Figure 5 we present an example of our case study: On the requirements level, we model subsystems as an inheritance hierarchy with one kind of subsystems having a buffer associated. This accurately reflects the situation in our case study where some subsystems have buffers, others do not. On the architecture level, we model the same concept using an optional association between a single subsystem class and the buffer class. The subsystem modeling in $OM_{requirements}$ facilitates the mapping process because the more detailed abstraction can be mapped more easily to the classes in OM_{design}.

2.4 OM_{target}

OM_{target} differs from all other object models in that it is semi-automatically created. The abstractions modeled as classes in OM_{design} are mapped to the oo high-level constructs in $OM_{requirements}$ to achieve an object-oriented target system. Human interaction is used in the mapping process to resolve conflicts and optimize the mapping result by introducing application and domain knowledge.

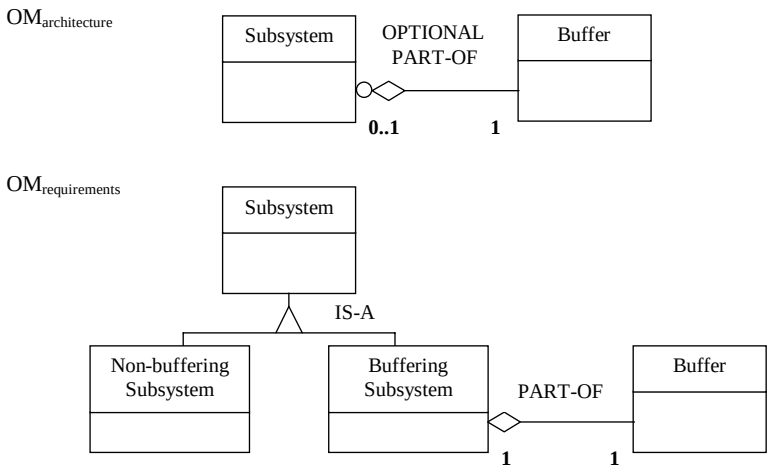

Figure 5: Abstraction differences in $OM_{requirements}$ and $OM_{architecture}$

OM_{target} is the basis for the system transformation. It represents an object model of the procedural system with the model entities linked to entities of the source code. In a code generation step, we use the procedural source code together with the information in OM_{target} to create the object-oriented target code.

3 Utilizing object models in abstraction-to-code mapping

In the object-oriented re-architecting process we face the concept assignment problem, i.e. the problem of relating concepts to code. We build abstractions of the underlying application concepts in a set of object models. The object-oriented paradigm provides various means to model these abstractions such as classes, associations, etc. Having the source code and a model of the application, we can start to map abstractions to code. Here, we face another problem concerning re-architecting: To get a re-architectured system of high quality in terms of object-orientation (i.e the extensive use of the oo mechanisms provided) we have to provide an appropriate and highly abstract target model. The higher the model abstraction level, the bigger the gap between code and abstraction and the poorer the mapping result.

3.1 Stepwise mapping

To bridge the gap between low-level code and high-level abstractions we introduce a stepwise mapping process. In the first step, the code is related to abstractions deduced mainly from design documentation and application domain knowledge. This OM_{design} object model lacks high-level oo concepts, in particular inheritance. To introduce high-level oo concepts we cre-

ate another model from the application requirements and domain knowledge. This $OM_{requirements}$ model has to represent exactly the system to be re-architectured but is not modeled strictly according to the structure of the procedural system. It rather is a structural representation of the target oo system. Explicit links between OM_{design} and $OM_{requirements}$ are established in the CORET mapping process finishing our model-driven abstraction-to-code mapping. The result is an object model with explicit references to the source code. These references are stored in a *mapping table* relating model entities to source code entities.
During the mapping process, OM_{target} is generated which serves as the basis for a later system transformation. The abstraction level of OM_{target} in the best case is $OM_{requirements}$, in the worst case OM_{design} (see Figure 2). Considering architectural reasoning or reasoning about system families $OM_{requirements}$ can further be abstracted to the $OM_{architecture}$ model being an architectural representation of the procedural system in oo terms.

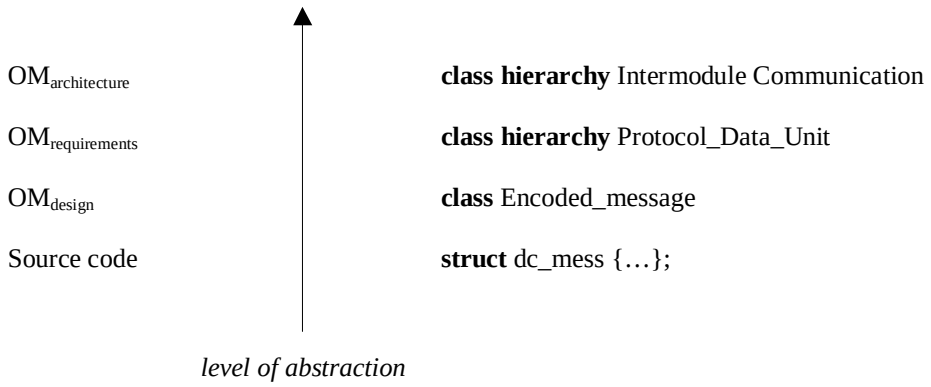

Figure 6: The 'message' concept at different levels of abstraction

3.2 Object model characteristics

As mentioned in Section 2 the object models are different in a number of aspects. The characteristic properties of the CORET object models that were described in Sections 2.1 to 2.4 are summarized in Table 1.
As an example, Figure 6 shows the different abstractions of the concept of a message. The object models provide the knowledge of the source code structure `dc_mess` being the part of the implementation of the high-level concept *Intermodule Communication*.
Obviously, the binding and mapping quality depend on the quality of the object models. If the design documentation is poor or not up to date, OM_{design} will considerably differ from the application's structure. In this case, having access to the knowledge of application experts is of great importance. If there are no explicit system requirements, they have to be re-engineered or extracted from the design documentation.

Properties	OM_{design}	$OM_{requirements}$
Information source	source code, design documentation, application expert	requirements documentation, application expert
Model abstraction level	low	high
Model granularity	high	low
Object-oriented concepts	no	limited
Relation to the legacy system	explicit links to source code accurate representation	explicit links to source code accurate representation
Relation to the target system	basis for target model	basis for target model
Model annotations	info source, class category	info source, class category
Model task	binding	mapping
Model creation mode	manually	manually
	$OM_{architecture}$	OM_{target}
Information source	requirements documentation, application expert	other models, application expert
Model abstraction level	high	Between OM_{design}
Model granularity	high	and $OM_{requirements}$.
Object-oriented concepts	full	limited
Relation to the legacy system	no explicit links to source code inaccurate representation	explicit links to source code accurate representation
Relation to the target system	–	basis for target model
Model annotations	info source	–
Model task	documentation	transformation
Model creation mode	manually	semi-automated

Table 1: Properties of the CORET object models

Human intervention is important in our OORA process. The CORET process is interactive, using application domain and application specific information to resolve uncertainties and ambiguities. Because of the process inherent uncertainty, we use concepts of soft-computing in the binding and mapping process, e.g. a fuzzy text comparison in our similarity measure [5].

4 Tools in the OORA process

To create and maintain the set of object models we use Rational's modeling tool *ROSE*. We use ROSE's ODBC enabled scripting language to export object model data to a RDBMS (Oracle 7[7]) and to import data to visualize and adapt models in ROSE. ROSE directly supports the textual annotation of each model entity. The Booch, OMT, and UML notations are supported.
For the identification of class candidates we used Imagix's reverse engineering tool *Imagix4D*. We extracted the main `struct`'s together with the procedures accessing and/or manipulating them. Since we plan to detect class candidates according to more sophisticated criteria, a dedicated CORET parser was built. Class candidates are as well stored in the RDBMS. The CORET binder works on the database to establish the OM_{desgin} – source code relation.
To support the human engineer we identify a couple of tasks that can be semi-automated and thus supported by tools (binding, mapping, etc.). Because OORA involves some uncertainties in each step, human interaction has proven to be useful and thus is essential throughout the whole process to resolve conflicts and ambiguities. Thus, we do not strive for full automation.

5 Further work

To improve the binding, we are going to introduce the so-called *concept annotation*. Model entities, especially methods, are annotated with concepts such as 'maximum search', 'sort', etc. In the reverse engineering step, program plans associated with these concepts are used to detect the concepts in the code. The concepts and plans together with their interrelations are stored in a pattern repository. These concepts and plans are functional and language patterns, respectively, we do not use (object-oriented) design patterns such as [7] or [10]. Since the binding process currently is based on a mainly syntactical similarity check and has deficiencies in associating methods and procedures, the concept annotation enables a semantical similarity comparison and thus is expected to improve the binding result.
Additionally, we plan to use language patterns to identify GUI (Graphical User Interface) and data structure implementations (arrays, lists, trees, etc.) in the procedural code. Since the GUI and data structure implementation is not involved with the basic application functionality, we consider to do the modeling in a separate object model.

[7] *Oracle* is a trademark of Oracle Inc.

6 Conclusion

In the OORA process, we attack the concept assignment problem by providing object models of the application at different levels of abstraction to enable a stepwise abstraction-to-code mapping process. As abstractions we use the class concept of the object-oriented paradigm. The re-architectured system uses the object-oriented mechanisms such as data abstraction, information hiding, and inheritance to improve the software evolution by facilitating the system maintainability or enabling software reuse. The whole process is based on human interaction to introduce and improve a variety of semantics that cannot be recovered by fully automated approaches.
The result of the abstraction-to-code mapping is a set of object models that have explicit references to the source code. This means that these models have a well-defined and formal represented corresponding to the source code rather than only being informal representations of the system. This is essential for maintenance since an engineer can then locate the system part subject to change in an application model and is guided to the corresponding parts in the code. The model-driven linkage of abstractions to code is a prerequisite in our OORA process to achieve an object-oriented system with high-level semantics.

7 Acknowledgments

We are grateful to Roland Mittermeir and Dominik Rauner-Reithmayer for many interesting discussions and valuable comments. This work, that was pursued in cooperation with the University Klagenfurt, was supported by the Austrian National Science Foundation (FWF), project no. P 11.340 ÖMA.

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. *Proceedings of the* 15^{th} *International Conference on Software Engineering, Baltimore, Maryland*, pages 482–98. IEEE Computer Society Press, May 1993.

[2] 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.

[3] M. Fowler and K. Scott. *UML distilled – applying the standard object modeling language*, Object technology series. Addison-Wesley, Reading, Mass. and London, 1997.

[4] H. Gall, R. Klösch, and R. Mittermeir. Object-oriented re-architecturing. *5^{th} European Software Engineering Conference (ESEC '95)*. Springer Verlag, Berlin, September 1995.

[5] H. Gall and J. Weidl. Binding object models to source code: an approach to object-oriented re-architecting. Technical report TUV–1841–97–14.

Distributed Systems Group, Technical University of Vienna, September 1997.

[6] H. C. Gall, R. R. Klösch, and R. T. Mittermeir. Using domain knowledge to improve reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, **6**(3):477–505. World Scientific Publishing Company, 1996.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass. and London, 1995.

[8] 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.

[9] 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.

[10] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*, ACM Press. Addison-Wesley, 1995.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[12] A. S. Yeh, D. R. Harris, and H. B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. *Second Working Conference on Reverse Engineering*, pages 227–36. IEEE Computer Society Press, July 1995.