Towards a Precise Description of Reverse Engineering Methods and Tools

Stan Jarzabek and Irene Woon
Dept. Information Systems & Computer Science,
National University of Singapore
{stan, iwoon}@iscs.nus.sg

Abstract
The potential and limitations of reverse engineering techniques is still a matter of debate and investigation. Both experimental studies and commonsense tell us that design abstractions are useful in program understanding and maintenance. In the case of incomplete program documentation, reverse engineering tools can recover some of the design abstractions from code. However, it is not clear which design abstractions can and which cannot be automatically recovered. This can be attributed to the understandable reluctance of industry to publicize explicit knowledge of this process due to its enormous commercial value and the fact that reverse engineering is a fairly new research discipline. As a start to formalizing what we already know about reverse engineering, we propose a framework for describing and evaluating reverse engineering methods and tools. First, we build design models for a source language and for the recovered design. Then, we describe what a given reverse engineering method or tool achieves as a formal mapping from the source language design model into the recovered design model. We show use object recovery scenarios to illustrate the presented concepts.

1. Introduction
The objective of reverse engineering [1] is to extract design information, functional specifications and, eventually, requirements from the program code, documents and any other sources available. Usually, reverse engineering is done in co-operation between human experts and a reverse engineering tool.

Reverse engineering techniques can help in maintenance of poorly documented programs and in software re-engineering projects. If, for example, we re-engineer a C program to C++, we would typically start by building an object model. Analysis and reverse engineering of existing data structures, file definitions and program code can help in building such a model. Experimental studies on program understanding [2,3] show that program comprehension involves creating multiple mental models of the software, identifying objects within each model, establishing how the objects interact within and across models. If those program models are not properly documented, programmers recover them from code and from other sources. Reverse engineering practice is, therefore, as old as programming. As a research discipline, reverse engineering is fairly new and the debate about reverse engineering objectives, potentials and levels of possible automation still continues [4,5,6]. Success stories with reverse engineering have been reported [7,8], many tools have been implemented both in industry [9,10,11] and academia [12,13,14,15,16,17], but more experimentation is needed to fully explore what reverse engineering tools can do, what they cannot do and how they should interact with programmers to make program understanding easier and program maintenance less expensive.

In order to study and evaluate any complex phenomenon, we must be able to describe it in a precise way. We felt that our understanding of the reverse engineering approach would be better if we could precisely say what a given reverse engineering method or tool does. In this paper, we describe a preliminary framework for specifying and evaluating capabilities of reverse engineering methods and tools.

2. A Proposed Framework
We describe reverse engineering methods and tools in terms of rules that a given method or tool uses to produce
higher level design abstractions from the low level ones (or just from a source code). We call these rules reverse engineering heuristics, a notion similar to a candidature criterion [17,18]. We view a reverse engineering process as application of reverse engineering heuristics. To apply a reverse engineering heuristic, we examine lower level program designs in order to find manifestations of higher level design concepts. The result of a reverse engineering process consists of two program models at different levels of abstraction and of a mapping between the models showing which elements in one model correspond to which elements in the other model.

We developed notations to specify well-understood reverse engineering heuristics and model program design at all levels of abstraction using extended OMT notation [19]. Then, we use an SQL-like language to specify design patterns and to describe how a higher level program model is to be created when certain patterns are found in the lower level program design model (Fig. 1).

![Diagram of higher and lower level program design models with mappings between them.]

**Fig. 1 Formalizing reverse engineering heuristics**

Our program design modeling notations are based on the formalism called Program Query Language, PQL for short. We developed PQL to formalize program queries [20]. We also based the design of a generic reverse engineering tool on the PQL interface [16].

3. Recovering Objects from C

In procedural programs, objects are implemented by data structures and fragments of procedural code that manipulate these data structures. In well designed programs, each of the object's operations can be implemented as a separate procedure. Identifying objects means finding clusters of data that represent object's state and procedures that implement object's operations. Sneed and Nyary [21] describe many types of objects that may be of interest in re-engineering data processing applications e.g. user interface objects, data objects, view objects.

It is possible to find objects by identifying global data structures that are likely to represent the state of some object or identifying dependencies between data types in a program [22], through the classification of data and behavioural abstractions [17], or by reverse engineering structure charts and data flow diagrams [23].

Suppose we are re-engineering C programs into C++ and we wish to identify candidate classes in C code. The process that leads to identifying classes involves applying heuristics (such as data coupling between C functions) and manual analysis of programs by a human expert. We assume that application domain and software engineering experts participate in the reverse engineering project. This may be one person playing two roles or two different people, depending on the situation. The domain expert is aware of application domain concepts that are candidates for classes, while the software engineering expert can judge which design and implementation concepts are candidates for sound and useful classes. Automatically identified classes will be presented to human experts for acceptance and refinement.

3.1 Typical Object Recovery Heuristics

Typical heuristics used in recovering object-like modules:

H-1. Any global data structure, say data-X, directly referred to (used or modified) by two or more functions should be considered a candidate data rep (or part of it) for some object X.

H-2. Any function that refers to data-X should be considered a candidate method for object X.

H-3. Any two candidate data reps, say data-X and data-Y, identified by applying rule H1, and such that there is a function that refers to both data-X and data-Y, are likely to form together a data rep for the same object. Candidate methods for this object are functions that refer to any of the data that forms the object's data rep.

H-4. Any user defined data type, say type X, that appears in the headings of two or more functions, such that the headings of those functions do not include any supertype of type-X, should be considered a candidate data rep (or part of it) for some class X.

H-5. Any function whose heading includes type-X should be considered a candidate method for class X.

H-6. Any two candidate data reps, say type-X and type-Y, identified by applying rule H4, and such that there is a function whose heading includes both type-X and
3.1.1 A plan for semi-automatic recovery of abstract objects as candidates for classes. To see how the above heuristics could be used, we shall outline a possible plan for semi-automatic recovery of candidate classes. First, we focus on abstract objects.

O-1. Apply rule H1 to find candidate data reps for objects.

O-2. Let the human expert select one of the candidate data reps, say data-X, and decide whether or not it forms a basis for a sound design or application domain object. To help in making the decision, the expert will view functions that refer to data-X (rule H2).

O-3. If the expert decides that there are not enough reasons to further evaluate data-X as a data rep for some object, he/she will repeat step O-2 for another data; else, the expert will proceed to step O-4.

O-4. Apply repeatedly rule H3 to find other data that, together with data-X, are likely to form a data rep for object X. To help in making the decision, the expert will view functions that refer to candidate data reps (rule H3).

O-5. At this point, the expert has already identified a set of data that, together with data-X, form a data rep for object X.

O-6. Find candidate methods for object X (rules H2 and H3). Let the human expert view and verify the candidate methods. This ends recovery of object X.

O-7. Repeat steps 2-6 until all the candidate data reps have been considered.

O-8. Map recovered objects to classes.

3.1.2 A plan for semi-automatic recovery of ADTs as candidates for classes.

A-1. Apply rule H4 to find user-defined types that may be candidates for ADTs.

A-2. Let the human expert select one of the candidate types, say type-X, and decide whether or not it forms a basis for a sound design or application domain ADT. To help in making the decision, the expert will view functions whose headings include type-X (rule H4).

A-3. If the expert decides that there are not enough reasons to further evaluate type-X as a candidate for ADT, he/she will repeat step A-2 for another data type; else, the expert will proceed to step A-4.

A-4. Apply repeatedly rule H6 to find other data types that, together with type-X, are likely to form an ADT X. To help in making the decision, the expert will view functions whose headings include data types under consideration (rule H6).

A-5. At this point, the expert already has identified a set of data types that, together with type-X, form an ADT X.

A-6. Find candidate methods for ADT X (rules H5 and H6). Let the human expert view and verify the candidate methods. This ends recovery of ADT X.

A-7. Repeat steps 2-6 until all the candidate type reps have been considered.

A-8. Map recovered ADTs to classes.

3.2 Source and Target Design Models

We start by modeling those aspects of C and C++ program designs that are relevant to object recovery. As notions of a class, method and data representation for objects are not present in the domain of C programs, we need to model objects independently of C programs. A PKB (Program Knowledge Base) is a repository that stores source program design and DKB (Design Knowledge Base) is a repository that stores reverse engineered design views (in our case, object models). Fig. 2 depicts conceptual models of the PKB and DKB.

![Fig. 2. Design models for C PKB and Object model DKB](image)

The diagrams of Fig. 2 show design entities (in rectangular boxes), their attributes (above the entity boxes) and entity relationships. A filled circle at the end of a relationship link stands for 'many' connectivity. The
meaning of a relationship link is clarified by the name attached to the link. Design entities are classified using ISA relationship (a triangle with general entity above and specialized entities below). Relationships defined for a parent entity (e.g., Cmodule) apply to child entities. Attributes assigned to a parent are inherited by all its children.

The reader will have noticed that we adopt a parsed, meaningful program representation (C PKB) rather than a C program text as a starting point for describing reverse engineering heuristics. This decision requires explanation. The actual contents of the PKB depends on the source language and on the class of reverse engineering heuristics we wish to describe. (For example, we created the C PKB model of Fig. 2 so that we can describe heuristics discussed in last section.) The PKB may contain abstract syntax trees annotated with semantic information, control and data flow relations, procedure call graphs, a database of program characteristics, a representation such as UNIFORM used in REDO project [13] or a hybrid representation that combines tree-like structures with a relational database of global program descriptions [24]. Furthermore, in the interactive reverse engineering and program analysis situation, it is more appropriate to precompute only essential program designs (such as syntax trees, control flow graphs and function call graphs) and compute other types of detailed information (such as data flow relations) on demand. We refer the reader to other publications for detailed discussion of possible media to store software engineering artifacts [10,25, 26].

We decided to clearly separate a conceptual model of program design stored in the PKB from the physical representation of program design in the PKB and from the actual mechanism used to compute program design. We express reverse engineering heuristics in terms of the conceptual program design models. Any type of program information that we need to describe reverse engineering heuristics and that can be computed from source programs using traditional compilation methods may be included into the PKB model. This arrangement simplifies descriptions of reverse engineering heuristics, as we can address reverse engineering problems in separation from routine source code level analysis tasks. This is not to say that computation of low level design abstractions does not pose its own problems. For example, computation of data flow relations in programs with pointers is difficult. In this paper, we do not address the issue of how low level design are computed. But we build program models carefully so that the PKB contains only the program information that is computable from source programs using conventional methods.

33 Specifying object recovery heuristics in PQL

In this section, we describe object recovery heuristics in our notations. We do not address here the issue whether and under what conditions the proposed reverse engineering heuristics yield sound objects. Our objective is merely to demonstrate how these and other reverse engineering heuristics can be described in a precise way.

With reference to the program models of Fig. 2, we shall now define heuristics H1-H6 previously discussed. In all the definitions, we will use names data, data-X and data1 as synonyms of design entityglobData and namefun as a synonym of design entity Cfunction. The following PQL declarations introduce synonyms:

globData data, data-X, data1
Cfunction fun

The definition of first-cut candidate objects (heuristics H1):
view objects
Select <data, fun> such that Refers (_, data>1 ) and RefersTo ( fun, data )

Explanation: Underscore means an unbound variable.

The definition of first-cut candidate methods (heuristics H2):
view methods
Select <fun> such that <_, fun> ∈ objects

The definition of first-cut candidate data reps for objects (heuristics H3):
view data-rep
Select <data> such that <data, _> ∈ objects

Explanation: This view is presented to the human expert in steps O-1 and O-2.

For a selected data-X from the set data-rep, we now produce first-cut methods for a candidate object X with data rep data-X (rule H2):
view method-X-1
Select <fun> from methods such that RefersTo ( fun, data-X )

Explanation: This view is presented to the human expert in steps O-2 and O-3.

view data-rep-X
Select <data> such that data = data-X

Explanation: data-X is judged to be a data rep for a sound object.
For a selected data1 from set data-rep-X, we find other data that, together with data-X, form a data rep for the same object:

```sql
view data-rep-X-more
Select < data > from data-rep
  such that data \in data-rep-X
  and such that exists [ fun in methods
    such that RefersTo ( fun, data1 )
    and RefersTo ( fun, data ) ]
```

*Explanation:* Each data from set data-rep-X-more is presented to the expert who will decide whether the data should be included into the set data-rep-X. This view is computed repeatedly as long as new data is contributed to the set data-rep-X (step O-4 and rule H3).

The alternative solution to computing the data rep for object X might be to compute all the candidate data reps, of object X and then ask the expert to review the candidates and decide ones that should collectively form a data rep for object X. Computing candidate data reps is formalized in the following recursive view:

```sql
recursive view data-rep-X
Select < data > from data-rep
  such that exists [ fun in methods
    such that data1 \in data-rep-X
    such that RefersTo ( fun, data1 )
    and RefersTo ( fun, data ) ]
```

*Explanation:* Here, we find other data that, together with data-X, possibly form a data rep for object X, based on rule H3. A recursive view is executed repeatedly as long as new data are inserted to the set data-rep-X. The reader can find out about strategies for evaluating recursive queries in [27]. Our current implementation does not include recursive views.

The definition of methods for object X (heuristics H3):

```sql
view methods-X
Select < fun > from methods
  such that exists [ data in data-rep-X such that RefersTo ( fun, data ) ]
```

*Explanation:* The expert reviews and verifies candidate methods (step O-6).

This completes recovery of design views relevant to object X as a candidate class for a C++ program. We do not formalize heuristics for recovery of ADTs as they are similar to those described above.

4. Generation of recovered design views in extended PQL

In the above discussion, we have recovered design views that correspond to elements in the object model but we did not create object models in an explicit way. We extended PQL to allow for generating design fragments using terminology of the target design, in the case of our example, Object-Oriented design. Generation rules have the following format:

```
source-pattern => ( CreateEnt entitySpec 1
CreateRel relSpec 1
AssignAttr attrValueSpec1
CreateLink linkedEntitiesSpec )*
```

The 'source-pattern' is a pattern definition in PQL. The right hand side of the formula shows how to create fragments of the reverse engineered target design when instances of the source-pattern are found in the PKB. Operator CreateEnt creates a new instance of a design entity, CreateRel establishes a relationship between two design entities, AssignAttr assigns a value to the entity or relationship attribute and CreateLink links design entities from different program design models for the purpose of traceability. Operators may appear any number of times and in any order on the right hand side of the formula.

In the object recovery example, the 'source-pattern' refers to C models (on the left hand side of Fig. 2), while target design entities are classes, attributes and methods of the Object-Oriented design model (on the right hand side of Fig. 2). In the extended PQL, we can map data from set data-rep-X to class attributes and functions from set methods-X to class methods:

```
CreateEnt class with class.className = ?X
```

*Explanation:* ?X indicates that the class name must be provided by the human expert.

```sql
Select fun from methods-X
  CreateEnt method with method méthName = fun.funName
CreateRel Declares ( class, method ) with class.className = ?X
CreateLink ( fun, method )
```

```sql
Select data from data-rep-X
  CreateEnt attribute with attribute.attrName = data.dataName
CreateRel Declares ( class, attribute ) with class.className = ?X
CreateLink ( data, attribute )
```

7
5. Related work

Other work that described design query languages include REFINE [10] and SCA [18]. REFINE’s query language is based on program models stored in an Object-Oriented database. SCA offers an algebraic query language. In [18], the authors describe a logic-based notation for specifying reverse engineering heuristics (called candidature criteria) and the program information required to isolate candidates. The above approaches differ one from another and from our notation in the underlying models of program design and in the way program queries are written. Unlike the other notations, our notation allows one to specify target designs that are not compatible with the source program design model.

The objective of the above efforts is twofold. On one hand, interpreters of design query languages provide programmer interfaces to reverse engineering, program analysis and program transformation tools. Such interfaces add much flexibility to tools as a programmer can modify and extend tool functions without re-programming the tool. On the other hand, the above notations allow us to precisely define program queries, reverse engineering heuristics and program transformations. They may enable us to study and evaluate capabilities of a wide class of tool used in software maintenance and re-engineering projects. It is unlikely that we may have one notation that will cover all heuristics, will be easy to understand for humans, will allow us to reason about heuristics, will facilitate efficient implementation, etc. It is not easy to compare different notations but it seems that REFINE’s language has the highest expressive power as it is Turing complete. Algebraic and logic notations have sound theoretical foundation and are suitable for reasoning. Our notation is intuitive and simple, based on SQL and Entity-Relationship query language concepts.

6. Conclusion

In this paper, we proposed a way of describing reverse engineering heuristics that is reasonably precise and intuitive. We used object recovery heuristics described by others to illustrate our program design modeling and reverse engineering heuristics specification method. In the last section, we reviewed notations based on other formalisms that were developed with similar purpose in mind. The message of the paper is that, with this tool box of notations, we might attempt to precisely describe new reverse engineering methods and tools proposed by both research community and industry. Apart from the practical value of such descriptions, strengths and weaknesses of different notations in different reverse engineering situations could be evaluated so that notations could evolve accordingly.

Conceptual program models and design query notations are powerful tool design instruments. In particular, we can express reverse engineering heuristics in a descriptive rather than in an operational way. We can also implement the same specifications of tool capabilities on variety of program representations to obtain required tool characteristics (performance, simplicity of the design, etc.). But this flexibility poses challenges for efficient evaluation of reverse engineering heuristics. One must take into account the choice of the storage media for the program knowledge, algorithms for computing the program knowledge from source programs and the issue of which program design information should be pre-computed by the reverse engineering front-end and which should be computed on demand during evaluation of reverse engineering heuristics. Efficient implementation of the PQL-based interpreter for reverse engineering heuristics is a hard problem that we try to address at both program representation and PQL design levels.

In future work, we plan to build more friendly user interfaces on top of the notation described in this paper. To apply design solutions presented in this paper to big industrial programs, we implement a hybrid PKB for efficient interpretation of reverse engineering heuristics. We also plan to experiment with more reverse engineering heuristics and refine our notations.

7. ACKNOWLEDGMENTS

This research was sponsored by the National University of Singapore Research Grant RP950616. Thanks are due to the following students who implemented various components of tools for program understanding: Tan Poh Keam implemented the data reverse engineering tool, Wang Guosheng implemented a prototype of the PQL interpreter and Ding Xin extended PQL with program transformation features.

REFERENCES

Integration and Professional Services, Vol. 1, IDC No. 3423 1-11.


