Reverse Engineering is Reverse Forward Engineering

Ira D. Baxter
Michael Mehlich

Semantic Designs, Inc.
12636 Research Blvd. C-214
Austin, TX 78759-2200
+1 512-250-1018

Abstract

Reverse Engineering is focused on the challenging task of understanding legacy program code without having suitable documentation. Using a transformational forward engineering perspective, we gain the insight that much of this difficulty is caused by design decisions made during system development. Such decisions "hide" the program functionality and performance requirements in the final system by applying repeated refinements through layers of abstraction, and information-spreading optimizations, both of which change representations and force single program entities to serve multiple purposes. To be able to reverse engineer, we essentially have to reverse these design decisions. Following the transformational approach we can use the transformations of a forward engineering methodology and apply them "backwards" to reverse engineer code to a more abstract specification. Since most existing code was not generated by transformational synthesis, this produces a plausible formal transformational design rather than the original authors' actual design. A byproduct of the transformational reverse engineering process is a design database for the program that then can be maintained to minimize the need for further reverse engineering during the remaining lifetime of the system. A consequence of this perspective is the belief that plan recognition methods are not sufficient for reverse engineering. As an example, a small fragment of a real-time operating system is reverse-engineered using this approach.

1. Introduction

Software engineering practice tends to focus on the design and implementation of a software product without considering its lifetime, usually 10 years or more [TAM92]. However, the major effort in software engineering organizations is spent after development ([Boe81, Gui83]) on maintaining the systems to remove existing errors and to adapt them to changed requirements.

Unfortunately, mature software systems often have incomplete, incorrect or even nonexistent design documentation. This makes it difficult to understand what the system is doing, why it is doing it, how the work is performed, and why it is coded that way. Consequently mature systems are hard to modify and the modifications are difficult to validate.

For newly developed systems the problem can be reduced by thoroughly documenting the system and maintaining the documentation together with the system. Ideally, the system documentation describes the product and the complete design, including its rationale.

Most captured design information is informal (not machine-interpretable). While this is valuable for the software maintainers, informal information is subject to wide interpretation, and so the usability is limited by the problem of different software developers using different interpretations of the same description.

Formal descriptions of the design (and its rationale) with a precise semantics can overcome this communication problem. It can even allow us to modify the design rather than the code (cf. [BAX95]) and, thus, to modify a software system using semiautomatic tools. Transformational development of software ([NEI84, PAR90]) guided by performance criteria ([MCC87]) seems to be the right way to get such descriptions of a design.

In such an ideal setting there would be no reason for reverse engineering. However, the large amount of existing software that has to be maintained forces us to face this problem.

Current reverse engineering technology focuses on regaining information by using analysis tools ([BM82] and by abstracting programs bottom-up by recognizing plans in the source code ([KNE92, RW90A, RW90B, TFAM96]). The major purpose of such tools essentially is to aid maintainers *understand* the program (cf. [RUG94]). While we believe in the utility of these approaches as part of a larger toolkit and vision, they are quite inadequate for the task of design recovery. Analysis tools, though valuable, only provide some design information derived from the structure of the code, not from its intention or construction. Pure plan recognition is unlikely to be powerful enough to drive reverse engineering for the following reasons:

- legacy code is written using rather tricky encodings to achieve better efficiency,
- all the necessary plan patterns in all the variations must be supplied in advance for a particular application, and
- different abstract concepts map to the same code within one application.

Even more important, the main reason for doing reverse engineering is to *modify* the software system; understanding it is only a necessary precondition to do so. We consider transformational development with modification of the design rather than the code as the means to accomplish incremental modification, as it provides us with considerable theory and techniques to carry it out ([BAX90A, BAX92A]). For existing systems, this implies a need to reconstruct a *plausible* transformational design that could have been used to derive the code from a suitable abstract specification of the system, i.e. we want to apply the transformations backwards from the program to its specification. Recording these transformations (together with the rationale for applying them) then allows us to modify the design instead of just the program code.

In the following we explore forward and reverse engineering in greater detail and provide an example for transformational reverse engineering legacy assembler code to a high-level concept.

2. Forward Engineering

In current forward engineering practice informal requirements are somehow converted into a *semi-formal* specification using domain notations without underlying precise semantics like e.g. data-flow diagrams, entity-relationship diagrams, natural language descriptions, or other problem specific informal or semiformal notations. The program then is constructed manually (i.e. in an error prone way) from the specification by a creative agent, the programmer.

Hidden in this creative construction of the program from the specification are a set of obvious as well as non-obvious design decisions about how to encode certain parts of the specification in an efficient way using available implementation mechanisms to achieve performance criteria (the why of the design decisions). As an example, a specification fragment requiring associative retrieval using numeric keys may be implemented using hash tables, achieving good system reaction time. These decisions are usually not documented.

Over time the program code is modified to remove errors and to adapt the system to changed requirements. The requirements may change to allow usage of alphanumeric keys and to be able to handle large amounts of data, and the implementation revised to use disk-based B-trees. Unfortunately, often these changes take place without being reflected correctly in the specification. The gap between the original specification and the program becomes larger and larger. The result is a program code without a proper specification and with untrustworthy design information (such as comments describing the hash tables!). The code becomes difficult to understand and, thus, difficult to maintain.

To overcome this deficiency, it is important to change the specification first and then reflect the changes in the program code. A necessary precondition for this is to have reliable information about the relationship between the specification and the program code. The design and its rationale describe the how and why of this relationship; however, they are not documented in current practice.

There are two known approaches to reduce the gap between the specification and the program.

The first one is the development of software by stepwise refinement introducing intermediate descriptions of the system between the specification and the final program code. The intermediate descriptions should reflect major design decisions during the construction, which helps to understand the software and its design. However, this approach has some important drawbacks: the development steps are still manual, they are too large not to contain hidden design decisions, there is no rationale directly associated to the steps, the relation between the descriptions can only be established a posteriori, and there are a lot more documents to be maintained for incorporating changes of the system requirements.

Despite these drawbacks, the stepwise refinement approach is not that bad. Consequently attacking them leads us to the second approach: the transformational development of software where the fairly large manual development steps are replaced by smaller formal correctness-preserving transformations each of which reflects a small design decision or optimization.

A necessary prerequisite for this approach is to have a *formal* specification which may contain domain notations (e.g. for the general purpose domains of logical descriptions and data-flow diagrams and for special application domains like e.g. money management and loans) by giving them a precise underlying semantics. Based on this the program code can then be derived by making small implementation steps using formal correctness-preserving transformations leading from more abstract specifications to more concrete specifications. The final program code then is *correct by construction* with respect to the formal requirements specification. Each transformation falls into one of three categories: refinements (i.e. maps of concepts from an abstract level to a more concrete one), optimizations (i.e. maps to reduce the resources used at a level of abstraction according to some criteria), and jittering transformations (that eventually enable the application of refinements and optimizations). The decision to apply a particular transformation is thus crucial design information. The applied transformations coupled with the rationale for the choice of them is the design information that "explains" the code. It is called the *transformational design* of the code from the specification.

The selection of a transformation to apply to a certain specification in general is still a creative process but is guided by the performance criteria to be achieved. This guided selection of the transformations allows the development process to be supported by a semiautomatic system that contains a large repertoire of available transforms by applying them to achieve or at least approach the performance criteria.

The transformations needed for practical software engineering comprise all those design decisions implicitly used by current software developers including techniques like the following ones (see e.g. [PAR90, RSW95, RUG94]):

- *Decomposition* - Most problems can be decomposed to subproblems which in general is possible in different ways. The actual hierarchical structure of the code represents only one particular choice from the set of possible decomposition.
- *Generalization/Specialization* - If different components are similar it may be possible to construct a more general version that comprises both of them as special cases. Vice versa, for efficiency reasons it may be a good choice to specialize a component for a certain (part of the) application.
- *Choice of representation* - To be able to implement high-level data types in program code it is often necessary to change its representation. A common example is the representation of sets by lists.
- *Choice of algorithm* - High-level concepts can be realized by an algorithm in many different ways. The choice between which may be enforced by badly documented or even undocumented performance criteria that have to be satisfied.
- *Interleaving* - For efficiency reasons it may be useful to realize different concepts in the same section of code or the same data structure.
- *Delocalization* - Certain high-level concepts may be spread throughout the whole code introducing distracting details in other concepts (see the study of the effects of delocalization on comprehension in [LS86]).
- *Resource Sharing* - Interleaved code often allows different concepts to share some resources like e.g. control conditions, intermediate data results, functions, names, and other computational resources.
- *Data caching/Memoization* - If some data, that has to be computed, is needed often or its computation is expensive it is worth to cache the data, i.e. to memoize it for later reuse.

- *Optimizations* - To achieve highly efficient code (due to the need of satisfying e.g. memory or reaction time constraints) many different optimizations are used. Such optimizations comprise
    - the folding and unfolding (inlining) program code,
    - the use of algebraic properties of functions,
    - the merging of computations by composing or combining functions,
    - partial evaluation (often in the form of context-dependent simplification),
    - finite differencing,
    - result accumulation,
    - recursion simplification and elimination,
    - loop fusion,
    - optimizations typically performed by good compilers (code motion, common subexpression elimination, partial redundancy removal), and
    - domain specific optimizations like e.g. minimizing a finite state machine recognizing a certain language.

All these design decisions may overlap and may be delocalized during the construction of the program code from the specification. Whether these methods are carried out mechanically by a tool or informally by smart programmers, the resulting software systems are very difficult to understand.

The transformational software development approach has the advantage that it allows the automatic recording of the design decisions made during the derivation of the final program code from the formal specification. Provided the selection of the transformations that are applied during this development are guided by the performance criteria to be achieved and the relation between the selection and the performance criteria is recorded we get the complete design together with its rationale as the product of software development. The code itself is just a byproduct that is correct by construction.

Recording this design information allows us to modify the specification instead of the code and then to modify the design to get a new implementation of the modified specification. The formal nature of transformations makes such design modifications supportable by a semiautomatic tool that increases the reliability and reduces the cost of maintaining the system.

3. Reverse Engineering

If all existing software had been developed using a transformation system recording the design and its rationale there would be no need for reverse engineering any code. However, we have to make a concession to reality. A huge amount of conventionally developed software exists and such systems will continue to be developed for a long time.

These systems have errors and continual demand for the enhancement of their functional and performance requirements. Modification is a necessity. This means that there is a fundamental need to understand them. As they are often badly designed and have an incomplete, nonexistent, or, even worse, wrong documentation without any design information, this is a challenging task.

Currently a major approach to understanding consists of trying to recognize plans (code fragments implementing programming concepts) bottom-up from the program to a more abstract description ([RW90A]). Essentially, this is done by pattern matching on an internal representation of the code (e.g. an abstract syntax tree) which leads to detecting patterns of higher-level plans or concepts in a lower-level code.

This approach has a major drawback: It depends on having developed all (or at least all important) plan instances that may occur in the code that we want to understand. Unfortunately these plans can be (and have been) realized in many different ways because of the interactions between many possible design decisions (i.e. transformations). Imagine a logical array of structures; it can be implemented as an array of structures or a structure of arrays. Code to sort the array will look very different depending of the choice of implementation. A set of plans for sorting must somehow account for all the possible different representations of the data.

Thus, to detect a plan in legacy code we have to be able to detect all these different realizations of the same plan. But unfortunately we need different patterns to detect these different realizations; and all these patterns have to exist in advance to be able to reverse engineer the code, which is impossible because of the multitude of conceivable realizations all solving the same problem.

Worse, we must take into account that the plans we want to recognize are interleaved, delocalized [LS86], and share resources. So we must find plan patterns in the code that are intertwined with other plan patterns both of which may be scattered throughout the whole program code.

Thus, to effectively perform the reverse engineering task we have to use some *creativity* to detect and separate the high-level concepts in the code; a task that in general cannot be performed in advance by providing a reasonable amount of predefined plan patterns even together with sophisticated pattern matching rules.

These problems can be overcome by two different approaches. The first one is to allow a reverse engineer to add further plans, realizing patterns, and possibly further intertwining rules[*] to enable a plan recognizer to detect the actual realization that can be found in the code. However, to be able to maintain the system we are not only interested in the plan that is realized in the code but also in the design and its rationale that could have led from the plan to its realization in the actual code. Thus, the reverse engineer not only has to provide the plan pattern for the actual realization of a plan but also the transformations and the justifications for applying them, i.e. (s)he has to forward engineer a piece of software *and* achieve a certain solution that has been used in the code.

```
Function ReverseEngineer(Code,SetOfTransforms) returns DesignHistory
begin
     // Assert: Transforms are of form LHS => RHS
     DesignHistory=empty; // no design information yet
     AbstractProgram:=Code; // starting state
     repeat
         { // Plan recognition component:

           MatchingTransformations:=
                   MatchRightHandSides(SetOfTransforms,AbstractCode);
           Display(MatchingTransformations);
         };
         { // Missing knowledge inserted here
            ChosenTransform:=
                  InteractivelyChoose(MatchingTransformations);
            If ChosenTransfrom=DONE
            Then Return DesignHistory;
            Else If ChosenTransform=NONE
            Then { ChosenTransform:=NewlyDefinedTransform();
                   SetOfTranforms:=
                        Union(SetOfTransforms,ChosenTransform);
                 };
             AbstractCode:=
               ApplyTransformInverse(ChosenTransform,AbstractCode);
             DesignHistory:=Append(DesignHistory,ChosenTransform);
         };
     end;
end
```

Figure 1: Simple Model of Transformational Reverse Engineering

As an alternative approach, we can achieve the same effect by using small transformations backwards (these transformations may or may not already be provided by a forward transformation engine) and abstract the code step by step the end result of which is an abstract plan (or a set of plans) corresponding to the given code. In essence, the transformations applied backwards "jitter" the code into a form in which canonical plans are more easily found.

[*] Note that the intertwining rules essentially are forward transformations. These transformations must be applied to existing patterns to derive new intertwined patterns.

Such an abstract plan may already exist in the system or may be new domain knowledge developed during the reverse engineering process. By recording the (reverse) transformation steps used to achieve that plan, we acquire a plausible design that can be used as a formal basis for maintaining the reverse engineered system by maintaining the design. Doing so then avoids the need for repeatedly reverse engineering the system over its lifetime, caused by the continuous modification of its code.

As a side-benefit the transformational reverse engineering approach lets us acquire domain knowledge as well as capture a possible design. It is not necessary to define the domain knowledge in advance. Samples include new concepts identified by the reverse engineer along with implementations as found in the code. This information can also be used to develop and maintain other systems; i.e. we gain additional knowledge that can be used by a forward engineer.

While we argue that transformational reverse engineering should be much more effective than pure plan recognition, though plan recognition has its place. First, the same pattern matching technology on which plan recognition is based is used by any rewrite component of transformation systems (see Figure 1). Second, plans may be able to "tile" a significant part of an existing system, providing strong clues about some aspects of the code. These clues can be used as cues for guiding the reverse application of transformations. As an example, finding a computation that maps a string to a natural number is a hint that hashing is occurring, and hash-table related transformations should be attempted.

Comparing transformational reverse engineering with transformational forward engineering we see that both need domain knowledge and a semiautomatic transformation engine that is guided by some additional criteria to reduce the effort needed to develop the product, i.e. the *system design*.

Thus, we can obviously conclude that:

Reverse engineering needs the same knowledge and infrastructure as forward engineering.

Reverse engineering "just" applies the forward transformations backwards, i.e. in reverse.

4. A Legacy Reverse Engineering Example: Semaphores

In this section we demonstrate that transformational reverse engineering is feasible in practice by applying it manually to a real example. Our purpose here is not to specifically carry out this task, nor to provide precise algorithms, but rather to highlight the various decisions and how to handle them transformationally. In particular, we want to make the point that plan recognition cannot handle this task, and that reverse transformation can. We do not claim that this process is automatic. We assume that the reverse engineer has considerable informal problem domain knowledge, and incrementally provides guidance to a tool that records the transformations used for later inspection and maintenance.

The example is hand-coded, highly tuned legacy Motorola 6809 assembler code from a 1980s operating system written by one of the authors. We will reverse engineer from the assembly source back to the abstraction the code represents. The domain knowledge used by the engineer is related to operating system code, specifically to semaphores. This small bit of information provides considerable focus to the process.

The goal of the code is to release a semaphore resource to allow another waiting, possibly higher priority task to eventually acquire the semaphore and continue.

Having some domain knowledge about task synchronization (including at least synchronization with semaphores) and not being interested in how the highest priority task gets the processor to run we expect to see something like the following abstract code:

```
procedure RT_UnLock(x: ptr to semaphore)
  begin            // unlock specified semaphore
        V(x);      // release a resource unit
        return;    // return to original caller
  end;
```

The actual code fragment we intend to reverse-engineer is provided below. (The reader need not examine it carefully, as we abstract it immediately on the next page). Our smart reverse engineer determines it is the right code by decoding the assembly labels, and doing some control flow analysis.

```
RT:UnLock ; unlock block of code whose semaphore is in (X)
          intds           ; lock out the world momentarily
          inc   scb:count,x ; anybody in queue ?
          bgt   RT:ITSX    ; b/ no, done releasing resource
          stx   itempx     ; save pointer to semaphore
          ldx   scb:tcbq,x ; pointer to TCB to activate
          ldd   tcb:nexttcb,x   ; find pointer to TCB following that
          stx   itempd     ; save pointer to TCB to activate
          ldx   itempx     ; pointer to semaphore
          std   scb:tcbq,x ; remove task from SCB queue
          ldx   itempd     ; pointer to TCB to make ready to run
RT:ITSC ; insert task at (X) into ready queue and switch contexts if needed
; Assert: interrupts are disabled here
          jsr   RT:ITIQ    ; insert task into ready queue
          ldx   RT:TCBQ    ; are we still highest priority task ?
          cmpx  RT:CTCB    ; ... ?
          beq   RT:ITSX    ; b/ yes, pass control to caller
          ldx   #RT:ISCH   ; no, force task switch
          jmp   RT:SInt    ; by interrupt to task scheduler
RT:ITSX   inten            ; enable interrupts and return to caller
          rts
```

The code comments here are original. (The comments in the transformed code are informally derived from these. We do not really expect a transformation system to do this.)

This code contains several design decisions some of which we consider in greater detail below. For this code, it is important to know that releasing a semaphore resource is interleaved with priority scheduling; for efficiency reasons the current task "voluntarily" gives up the processor provided it is no longer the highest priority task. Thus, in order to be able to reverse engineer this code we must have (or develop) some knowledge about priority scheduling (a problem that goes beyond the scope of this paper).

Since the gap between the assembler code and the procedural program code we want to abstract to is pretty large we make a first step back using an intermediate domain notation that is sometimes used by compilers to translate from higher-level procedural code to assembler: a register transfer language (RTL). To do that we can apply the following straightforward **`TransformSet`** of refining transformations *backwards*:

```
call subroutine;                        ⇒ jsr  subroutine
goto target;                            ⇒ jmp  target
return;                                 ⇒ rts
goto x+offset;                          ⇒ jmp  offset,x
x:=address(variable);                   ⇒ ldx  #variable
x:=variable;                            ⇒ ldx  variable
variable:=x;                            ⇒ stx  variable
x:=x->offset;                           ⇒ ldx  offset,x
d:=x->offset;                           ⇒ ldd  offset,x
(x->offset):=d;                         ⇒ std  offset,x
int_enable:=true;                       ⇒ inten
int_enable:=false;                      ⇒ intds
(x->offset):=x->offset+1; cc:=compare(x->offset,0); ⇒     inc    offset,x
cc:=compare(x,variable);                ⇒ cmpx variable
if cc=0 then goto label fi;             ⇒ beq  label
if cc>0 then goto label fi;             ⇒ bgt  label
```

These transformations are M6809 specific. Obviously, these transformations are simple pattern matching rules that can be applied to existing assembler code simply and automatically. The design history must capture the application of all of these; it can capture them as a set rather than a linear sequence since there is no real order requirement.

Before looking at the resulting RTL-code for our example let us introduce four simple optimizing transformations (which are equivalence transformations and, thus, may be used in both directions for our purpose):

[common subexpression elimination]

`code(expression) ⇔ variable:=expression; code(variable)`

where `variable` is a new variable not occurring anywhere else in the program code and there is no assignment to a variable occurring in `expression` in the program code `code`

[register life-time renaming]

```
newvariable:=expression1; code(newvariable); variable:=expression2;
   ⇔ variable:=expression1; code(variable); variable:=expression2;
```

where `newvariable` is a new variable not occurring anywhere else in the program code

[dead assignment elimination]

```
variable:=expression; ⇔ ε
```

if `variable` is not used again in the code execution before a value is assigned to it the next time

[compare introduction]

```
(expression1 relation expression2)
   ⇔ compare(expression1,expression2) relation 0
```

These transformations are useful for most assembly reverse engineering tasks. We remark that the simple model of structural pattern matching used by most plan recognition engines does not work for these; data flow analysis is required.

Applying the refining and optimizing transformations backwards may lead us to the following RTL-code:

```
RT_UnLock: // unlock block of code whose semaphore is in register X
           int_enable:=false; // lock out the world momentarily
           (x->scb_count):=(x->scb_count)+1;     // release resource unit
           if (x->scb_count>0) then goto RT_ITSX fi;
           ready_task:=(x->scb_tcbq);          // pointer to TCB to activate
           (x->scb_tcbq):=
                   ready_task->tcb:nexttcb;    // remove task from SCB queue
           x:=ready_task;              // pointer to TCB to make ready to run
RT_ITSC:   // insert task X into queue and switch contexts if needed
           call RT_ITIQ;                       // insert task into ready queue
           if (RT_TCBQ=RT_CTCB)
           then goto RT_ITSX fi;               // still highest priority?
           x:=address(RT_ISCH);                // no, force task switch
           goto RT_SInt;                       // by interrupt to scheduler
RT_ITSX:   int_enable:=true;                   // enable interrupts
           return;                             // return to caller
```

To be able to reverse engineer further we use our domain knowledge to detect some non-obvious optimizations that have been used during the development of the code:

- The statement "**goto** RT_SInt;" actually does not indicate a continuation of "RT_ITSC" but is an optimized call of a subroutine with a following "**return**;". Thus it has to be replaced by the statements "**call** RT_SInt; **return**;".
- "RT_ITSC" itself is a subroutine that happened to be located at the end of "RT_UnLock" but may be called from another place in the program code (which is not shown in this program fragment). This means that we have to replace the control flow from "RT_UnLock" to "RT_ITSC", that is caused by incrementing the program counter in the processor, by a call to "RT_ITSC" as a subroutine followed by a "**return**;".
- "RT_ITSX" is a continuation for both "RT_UnLock" and "RT_ITSC" under certain conditions. It is a parameterless subroutine for its own. Thus jumps to it (using "**goto** RT_ITSX;") have to be replaced by "**call** RT_ITSX; **return**;" For sake of simplicity we inline the simple code of this procedure into the caller.
- There is a hidden convention in the code for the parameters to the procedures "RT_UnLock" and "RT_ITSC" (and "RT_SInt") which are placed in the register X. To be able to reverse engineer to parametric procedures we have to detect this fact. Then we can replace "*label*: *code* **return**;" by "**procedure** *label*(x) **begin** *code* **return**; **end**;" provided there is no **goto**-statement in the whole program code entering or leaving the body of the procedure, i.e. *code*. This includes a **goto**-statement to "*label*". Correspondingly, for these procedures we replace "**call** *label*;" on the caller sides by "**call** *label*(x);".

Additionally using the obvious equivalence transformation

[if-then-else with procedure return]

```
  if condition then code1; return; fi; code2; return;
     ⬄ if condition then code1; else code2; fi; return;
```

if there is no **goto**-statement entering or leaving *code1* or *code2*

and some of the optimizing transformations given above we get the following program code:

```
procedure RT_UnLock(x)
  begin // unlock block of code whose semaphore is in register X
       int_enable:=false;              // lock out the world momentarily
       (x->scb_count):=(x->scb_count)+1; // release resource unit
       if (x->scb_count>0) then        // no waiting task
         int_enable:=true;             // enable interrupts
       else                            // waiting task
         ready_task:=(x->scb_tcbq);    // pointer to TCB to activate
         (x->scb_tcbq):=ready_task->tcb:nexttcb; // remove task from SCB queue
         call RT_ITSC(ready_task);
       fi;
       return;
  end;
procedure RT_ITSC(x)
  begin // insert task X into queue and switch contexts if needed
       call RT_ITIQ(x)                 // insert task into ready queue
       if (RT_TCBQ=RT_CTCB) then       // we are still highest-priority task
         int_enable:=true;             // enable interrupts
       else                            // another task is highest-priority
         call RT_SInt(address(RT_ISCH)); // force task switch by scheduler call
       fi;
       return;
  end;
```

Now let us try to gain a deeper insight into the procedure “RT_ITSC”. To do that we have to know something about the organization of the ready queue (which is not possible by just looking at the program fragment we can handle in this paper):

The tasks in the ready queue are ordered with respect to their priority, i.e. the highest priority task is at the beginning of the queue whereas the lowest priority task can be find at the end of the queue. The pointer to the whole queue is identical to the pointer to the first task in the queue.

Looking into the procedure “RT_ITSC” we see a check whether the current task ‘RT_CTCB” is the same as the highest-priority task at the head of the ready queue “RT_TCBQ”. Thus, the task scheduler (which switches the program context to execute the highest-priority task) is called if and only if the highest priority task is not the current task (which called the procedure). This means that the solution reflects the fact that a context switch to another task is expensive and therefore should be avoided if the other task is just the current one. However, we are not interested in getting a highly efficient program code but in getting an abstract description of the procedure, i.e. we do not have to avoid the call to the scheduler for efficiency reasons. With the additional knowledge of “int_enable=true” after calling “RT_SInt” we can simplify “RT_ITSC” to:

```
procedure RT_ITSC(x)
  begin // insert task X into queue and switch contexts if needed
       call RT_ITIQ(x)            // insert task into ready queue
       call RT_SInt(address(RT_ISCH)); // force task switch by scheduler call
                                  // current task may get the processor
       return;
  end;
```

Inlining[*] the procedure “RT_ITSC” into “RT_Unlock” then allows us to move the statement “int_enable:=true;” from inside the **if**-statement to the end of the procedure.[&] Performing some further equivalence transformations on the **if**-statement and its condition we obtain the following code:

[*] This inlining does not mean that the procedure “RT_ITSC” becomes obsolete. The procedure is called from additional places in the operating system.

[&] Note, that “int_enable=true” is valid after calling “RT_SInt” which means that we can add the statement “int_enable:=true;” directly following the call without changing the meaning of the program code.

```
procedure RT_UnLock(x)
  begin // unlock block of code whose semaphore is in register X
       int_enable:=false;              // lock out the world momentarily
       (x->scb_count):=(x->scb_count)+1; // release resource unit
       if (x->scb_count<=0) then       // waiting task
         ready_task:=(x->scb_tcbq);     // pointer to TCB to activate
         (x->scb_tcbq):=ready_task->tcb:nexttcb; // remove task from SCB queue
         call RT_ITIQ(x)                // insert task into ready queue
         call RT_SInt(address(RT_ISCH)); // force task switch by scheduler call
       fi;
       int_enable:=true;               // enable interrupts
       return;
  end;
```

Because of lack of space in this paper we cannot present the details of further reverse engineering of this code. However, the next step has to recover the abstract data types used, i.e. “semaphore” and “task” together with their associated procedures. Doing that finally could lead us to the following code:

```
procedure RT_UnLock(x: ptr to semaphore)
  begin // unlock specified semaphore
       int_enable:=false;              // lock out the world momentarily
       call Semaphore.Release(x);      // release resource unit
       if (Semaphore.Desired(x)) then  // waiting task
           // insert task into ready queue:
           call Task.InsertIntoReadyQueue(Semaphore.Remove(x))
           // continue highest-priority task:
          call Task.ContinueHighestPriorityTask();
       fi;
       int_enable:=true;               // enable interrupts
       return;
  end;
```

Abstracting this in a task synchronization domain then should lead us to the expected abstract code:

```
procedure RT_UnLock(x: ptr to semaphore)
  begin            // unlock specified semaphore
       V(x);       // release a resource unit
       return;     // return to original caller
  end;
```

Recording all the transformations used in this process allows us to reapply them to this code to reproduce the original code. We note that replacing the transformations that intertwine priority scheduling code with spin-lock code will allow us to regenerate a revised semaphore unlock procedure for multiple processors. This is precisely the kind of maintenance task that we desire reverse engineering to support.

Now let us shortly consider a big problem that may arise when trying to reverse engineer the code using plan patterns and look specifically at the priority scheduling.

Let us assume that there exist some domain knowledge and associated plan patterns about priority scheduling. On an abstract level we may find e.g. the following procedural plan for adding an additional task into the ready queue and (re)scheduling the tasks:

```
Task.InsertIntoReadyQueue(task, priority);
Task.RunHighestPriorityTask();
```

However, in the assembler code we found something that roughly corresponds to the following refined plan (that demands the usage of a certain class of implementations for priority queues):

```
Task.InsertIntoReadyQueue(task, priority);
if Task.Current()=Task.FirstTaskInReadyQueue() then
  Task.ContinueCurrent();
else
  Task.ContinueHighestPriorityTask();
fi;
```

Who produced/produces this refined plan? Can we expect it to be part of an existing domain about priority scheduling? Which additional refinements of the original plan do we need to cover solutions used in other legacy codes for priority scheduling?

The answer to these questions is: No one can provide in advance all the possible plan refinements that may be found in legacy code. This means that we need a creative plan refinement generator, i.e. a semiautomatic forward transformation engine. But then, why don't we just generate the plan refinement backwards (on demand) by reverse engineering the code using the forward transformations in the reverse direction?

5. Related Work

Considerable effort has been invested in plan recognition. The Programmer's Apprentice ([RW90A]) is still one of the most sophisticated, representing plans ("cliches") using data and control flow. This representation provides some language independence and maximizes the ability to match a cliché against arbitrarily ordered code. PA essentially attempts to tile (with overlaps) a program recursively with cliches to explain it. We hope our example convincingly shows that recursively tiling cannot explain even small fragments of code.

A difficult problem with a large set of plans is the cost of matching them to a potentially very large program. Efficient graph matchers ([RW90B]) and constraint satisfaction solvers ([WQ95]) have been implemented to help match plans against code. Both these types of matchers can be used to good effect in a transformation system. However, we hope to avoid the "big-bang" problem of explaining the entire program at once by performing incremental recovery, storing the results of previous sessions in design histories, and reusing the design for maintenance.

The Maintainer's Assistant (MA) [WB96] attacks the software maintenance problem by using transformational methods, using a core representation for programs derived from the lambda calculus. This gives an excellent foundation with clear semantics, and so allows serious transformations to be applied to the original code. MA appears to be used for porting and translation rather than explanation and modification.

6. Concluding Remarks

The reason for performing reverse engineering is to maintain legacy code. Therefore, it should not be focused on program understanding but on system maintenance instead. This should be done in a way that frees us from reverse engineering a system again and again because of modifications made to its code over time.

Hence, reverse engineering has to be focused on design recovery. As we need sophisticated tool support to perform system maintenance efficiently, the recovered design has to be based on formal methods.

We have seen that transformational reverse engineering is a feasible approach to recover such a design. This means that we essentially need the same domain knowledge and infrastructure as we need for transformational forward engineering that can provide us with a design during the systems' construction. All the domains, their notations, and the associated transformations within each domain as well as between domains that are needed for forward engineering are also needed to reverse engineer legacy code to recover its missing design.

Transformational reverse engineering can easily be performed incrementally which allows us to reverse engineer only those parts of the code that are needed for an actual maintenance task. Over time more and more parts of the code will be abstracted and the abstraction level will become higher and higher possibly until we get an abstract requirements specification of the whole system.

All this can be supported effectively by a domain based transformation engine that can apply transformations forwards and backwards and can modify a design incrementally. Such a system, called the Design Maintenance System (DMS), which is intended for handling commercial size software, is currently under development at Semantic Designs. We hope to be able to report on concrete experience next year.

References

[BAX90] I.D. Baxter. *Transformational Maintenance by Reuse of Design Histories*. PhD thesis, University of California at Irvine, 1990.

[BAX92] I.D. Baxter. *Design Maintenance Systems*. Communications of the ACM 35(4):73-89, 1992.

[BAX95] I.D. Baxter. *Design (Not Code!) Maintenance*. In: ICSE-17 Workshop on Program Transformation for Software Evolution, 1995.

[BM82] V.R. Basili and H.D. Mills. *Understanding and Documenting Programs*. IEEE Transactions on Software Engineering 8(3):270-283, 1982.

[BOE81] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[GUI83] T. Guimaraes. *Managing Application Program Maintenance Expenditures*. Communications of the ACM 26(10):739-746, 1983.

[KNE92] V. Kozaczynski, J.Q. Ning, and A. Engberts. *Program Concept Recognition and Transformation*. IEEE Transactions on Software Engineering 18(12):1065-1075, 1992.

[LS86] S. Letovsky and E. Soloway. *Delocalized Plans and Program Comprehension*. IEEE Software 3(3):41-49, 1986.

[MCC87] R.D. McCartney, *Synthesizing Algorithms with Performance Constraints*, Ph.D. thesis, Brown University, 1987.

[NEI84] J.M. Neighbors. *The Draco Approach to Constructing Software from Components.* IEEE Transactions on Software Engineering 10(5):564-574, 1984.

[PAR90] H.A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[RSW95] S. Rugaber, K. Stirewalt, and L. Wills. *The Interleaving Problem in Program Understanding*. In: Working Conference on Reverse Engineering, pp. 166-175, 1995.

[RUG94] S. Rugaber. *White Paper on Reverse Engineering*. Georgia Institute of Technology, 1994.

[RW90A] C. Rich and R.C. Waters. *The Programmer's Apprentice*. ACM Press, 1990.

[RW90B] C. Rich and L. Wills. *Recognizing a Program's Design: A Graph Parsing Approach.* IEEE Software 7(1):82-89, 1990.

[TAM92] Tetsuo Tamai and Yohsuke Torimitsu, "Software Lifetime and its Evolution Process over Generations", *Proceedings of 1992 Conference on Software Maintenance*, November 1992, IEEE Computer Society Press.

[TFAM96] P. Tonella and R. Fiutem and G. Antoniol and E. Merlo. *Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic - A Case Study*. In: Working Conference on Reverse Engineering, pp. 198-207, 1996.

[WB96] M. P. Ward and K.H. Bennett *Formal Methods for Legacy Systems*. Journal of Software Maintenance: Research and Practice, 7(3):203-220, 1995.

[WQ95] S. Woods and A. Quilici. *Some Experiments Toward Understanding How Program Plan Recognition Algorithms Scale.* In: Working Conference on Reverse Engineering, pp. 21-30, 1996.