

A Two-Phase Approach to Reverse Engineering Using Formal Methods

Gerald C. Gannod and Betty H. C. Cheng

Computer Science Department
Michigan State University
East Lansing, Michigan 48824
USA

Abstract. Reverse engineering of program code is the process of constructing a higher level abstraction of an implementation in order to facilitate the understanding of a system that may be in a “legacy” or “geriatric” state. Changing architectures and improvements in programming methods, including formal methods in software development and object-oriented programming, have prompted a need to reverse engineer and re-engineer program code. This paper presents a two-phase approach to reverse engineering, the results of which can be used to guide the re-implementation of an object-oriented version of the system. The first phase abstracts formal specifications from program code, while the second phase constructs candidate objects from the formal specifications obtained from the first phase.

1 Introduction

Software maintenance has long been a problem facing software professionals, where the average age of software is between 10 to 15 years old [19]. With the development of new architectures and improvements in programming methods and languages, including formal methods in software development and object-oriented programming, there is a strong motivation to reverse engineer and re-engineer existing program code in order to preserve functionality, while exploiting the latest technology.

Reverse engineering of program code is the process of constructing a higher level abstraction of an implementation in order to facilitate the understanding of a system that may be in a “legacy” or “geriatric” state. Re-engineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [7]. The benefits offered by re-engineering versus developing software from the original requirements is considered to be a solution for handling legacy code because much of the functionality of the existing software has been achieved over a period of time and must be preserved for many reasons, including providing continuity to current users of the software [2].

One of the most difficult aspects of re-engineering is the recognition of the functionality of existing programs. This step in re-engineering is known as reverse engineering. Identifying design decisions, intended use, and domain specific details are often the main obstacles to successfully re-engineering a system.

The following terms are frequently used in the discussion of re-engineering [7].

Forward Engineering: The process of developing a system by moving from high level abstract specifications to detailed, implementation-specific manifestations.

Reverse Engineering: The process of analyzing a system in order to identify system components, component relationships, and intended behavior. These representations are then used to create higher level abstractions of the system.

Restructuring: The process of creating a logically equivalent system at the same level of abstraction. This process does not require semantic understanding of the system and is best characterized by the act of transforming “spaghetti” (unstructured) code into structured code.

Re-Engineering The examination and alteration of a system to reconstitute it in a new form and the subsequent implementation of the new form.

A diagram depicting the relationships between each of the terms is shown in Figure 1.

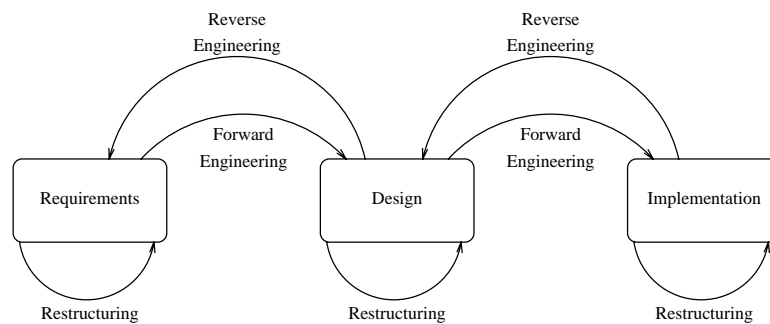


Fig. 1. Relationship between re-engineering terms

Common reverse engineering methods used by software maintenance engineers are observation (i.e., test case analysis) and examination of source code. These techniques are often tedious and error-prone.

Formal methods in software development provide many benefits in the forward engineering aspect of software development [22, 9, 17, 14, 11]. One of the advantages to using formal methods in software development is that the formal notations are precise, verifiable, and facilitate automated processing [3]. For any specification, there may be several implementations that satisfy the specification [10]. There have been recent investigations into reverse engineering that focus on the use of rigorous mathematical methods for extracting formal specifications from existing code [4, 20, 16]. Specifically, Lano and Breuer developed a framework for abstracting Z specifications from programs [16]. This approach involves the translation of imperative programming constructs into a higher level mathematical representation based on category and monad theory. Liu and Wilde have proposed an approach to identifying objects in procedural code [18], where the characterization of candidate objects is based on recognizing common routines, operations, data types, and data items through the examination of global data and major data types. Finally, Haughton and Lano have investigated the identification of objects in procedural code as well as specifications [12]. Recognition is based on translation to an intermediate language, program

slicing and heuristic identification of objects.

This paper presents a two-phase approach to reverse engineering that integrates a process for abstracting formal specifications from program code with a technique for identifying candidate objects in program code. One of the difficulties in automating the abstraction of a formal specification from program code is that the specification can often be too tightly bound to the implementation. Ultimately, this coupling necessitates user interaction in order to correctly obtain an accurate high level specification that is free of implementation bias. By taking full advantage of the logical properties of programming constructs, a precise determination of a program's purpose can be represented at a higher level of abstraction, as compared to program code. The constructed formal specification, along with information provided by a domain expert (someone who is knowledgeable about the specific domain, implementation details, and functionality requirements), facilitates determining program correctness using automated reasoning techniques.

Phase 1 of the approach is the process of abstracting formal specifications from program code. In earlier investigations, the AUTOSPEC tool [4] was developed to support the Phase 1 techniques for translating basic programming constructs into equivalent formal representations [4]. In order to illustrate the abstraction process, consider the following sequence of Pascal code.

```
for i := 0 to n
  if a[i] <= a[i+1]
    then
      m := a[i+1]
```

We can reverse engineer this sequence of code to obtain a formal specification that states that variable m is greater than all elements in the array a . Notationally, this specification is expressed as

$$(\forall i : 0 \leq i \leq n : m \geq a[i]).$$

This specification can be used to guide the re-implementation of the original routine in another language [6].

The use of object-oriented analysis, design, and programming has grown significantly in popularity over recent years [8]. Information hiding, abstraction, dynamic binding, and inheritance have made the paradigm appealing. As a result, there is a growing demand for transforming existing software into object-oriented software. The second phase of the approach provides a framework for identifying candidate objects in a system. Information found in formal specifications constructed by Phase 1 provides essential details that are used to identify candidate objects embedded in the implementation of a system. Reverse engineering program code into a representation that identifies candidate objects can provide a view of the subject system that facilitates the forward engineering of the design into an object-oriented implementation.

The remainder of this paper is organized as follows. Section 2 describes the techniques used to abstract formal specifications from program code. The process used to determine candidate objects in formal specifications is discussed in Section 3. Section 4 gives concluding remarks and briefly discusses future investigations.

2 Translation of Source Code into Formal Specifications

Reverse engineering program code into a formal specification facilitates the utilization of the benefits of formal methods for the maintenance and re-implementation of legacy code [4]. This section describes a translational approach that uses the logical properties of programs to abstract formal specifications from program code.

2.1 Review of Formal Methods

A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. For a given postcondition R and a statement S , the *weakest precondition* $wp(S, R)$ describes the set of states in which the statement S can begin execution and terminate with R true [10].

2.2 Abstraction of Formal Specifications from Programs

This section describes the techniques used to abstract formal specifications of programming constructs (i.e., assignments, alternatives, iteratives, and procedures).

Assignments. Given an assignment statement of the form $\mathbf{x} := \mathbf{e};$ and a precondition U , where U is a logical expression, the following annotated code is constructed

```

{U}                /* precondition */
x := e;
{x = e ∧ U}        /* postcondition */

```

The wp of an assignment statement is expressed as $wp(\mathbf{x} := \mathbf{expr}, R) = R_{\mathbf{expr}}^{\mathbf{x}}$, which represents the postcondition R with every occurrence of x replaced by the expression \mathbf{expr} . This type of replacement is termed a textual substitution of x by \mathbf{expr} in expression R . If x corresponds to a vector \bar{y} of variables and \mathbf{expr} represents a vector \bar{E} of expressions, then the wp of the assignment is of the form $R_{\bar{E}}^{\bar{y}}$, where each y_i is replaced by E_i , respectively, in expression R . In the above example, the wp of the assignment is U , which is satisfied by the original precondition.

Alternatives. Given an alternative statement of the form

```

if
    B1 → S1;
    B2 → S2;
    ...
    Bn → Sn;
fi;

```

and a precondition U , where $B_i \rightarrow S_i$ is a guarded command, and statement S_i is executed only if the boolean expression (guard) B_i is *true*, then a specification is constructed of the form

$$((B_1 \wedge post(S_1)) \vee \dots \vee (B_n \wedge post(S_n))) \wedge U,$$

where $post(S_i)$ is the postcondition of statement S_i . For each guarded command, one disjunct is abstracted, where the guard B_i is directly included as part of the disjunct. The abstraction algorithms are applied to statement list S_i in order to obtain the remaining logical expressions for the disjunct, $B_i \wedge post(S_i)$.

The *wp* of the alternative statement is: at least one guard B_i must be *true* and every guard must logically imply the *wp* of its corresponding statement list S_i with respect to the postcondition R [10]. Symbolically, the *wp* is expressed as

$$(\exists i :: B_i) \wedge (\forall i :: B_i \rightarrow wp(S_i, R)),$$

where ‘ $::$ ’ indicates that the range of the quantified variable i is not used in the current context.

Iteratives. An iterative statement takes the form

$$\begin{array}{l} do \\ \quad B_1 \rightarrow S_1; \\ \quad B_2 \rightarrow S_2; \\ \quad \dots \\ \quad B_n \rightarrow S_n; \\ od; \end{array}$$

Gries defines a number of guidelines for developing loops through the identification of loop invariants [10]. The methods of *deleting a conjunct*, *replacing a constant by a variable*, *enlarging the range of a variable*, and *adding a disjunct* can provide insight into the automated construction of a specification from program code. For instance, a loop written using the method of *replacing a constant by a variable* has properties that allow for the identification of the upper (lower) bound of an incremented (decremented) variable. Furthermore, determining the statements that ensure progress towards termination is facilitated by the properties associated with this class of loops. Figure 2 gives the steps for constructing a specification for a loop that was developed using the replace a constant by a variable strategy for the loop invariant.

In the cases where no automated heuristic can be applied to a loop or the constructed specification is incorrect, the domain expert is prompted for the proper specification of the statement. The following items are then identified in order to confirm that the specification of the loop is complete:

- *invariant (P)*: an expression describing the conditions prior to entry and upon exit of the iterative structure.
- *guards (B)*: Boolean expressions that restrict the entry into the loop. Execution of each guarded command, $B_i \rightarrow S_i$ terminates with P *true*, so that P is an invariant of the loop.

$$\{P \wedge B_i\}S_i\{P\}, \text{ for } 1 \leq i \leq n$$

When none of the guards is *true* and the invariant is *true*, then the postcondition of the loop should be satisfied ($P \wedge \neg BB \rightarrow R$, where $BB = B_1 \vee \dots \vee B_n$ and R is the postcondition).

- *bound function* (t): an integer expression representing the bound on the number of iterations. If at least one of the guards is true and the invariant is true, then the number of iterations is bounded below by t ($P \wedge BB \rightarrow (t > 0)$).
- *statements that make progress towards termination* (S_i): these statements must decrease the bound function after each iteration. Each loop iteration is guaranteed to decrease the bound function. Formally, this condition is:

$$\{P \wedge B_i\}t_1 := t; S_i\{t < t_1\}, \text{ for } 1 \leq i \leq n,$$

where $P \wedge B_i$ indicates that the invariant P and guard B_i are *true*, the assignment to t_1 represents the statement making progress towards termination, and S_i represents statements necessary to ensure that the invariant P is still *true* after one iteration.

1. The abstraction algorithm begins with the template for a quantified expression of the form

$$(Q i : range(i) : expression(i)),$$

where Q represents one of the quantifier symbols \forall, \exists, Σ .

2. The quantified variable(s) are determined by examining the identifiers occurring in guards B_j .
3. The ranges of the quantified variables are determined by finding statements occurring prior to entry into the loop that assign values to incremented (decremented) variables and their occurrences in the guards.
4. For each guarded command, the corresponding statement list includes statements that ensure progress towards termination; the postcondition for the remaining statements constitute $expression(i)$.
5. The bound function becomes the difference between the upper (lower) bound for a variable that is being incremented (decremented) and its value during loop iterations.

Fig. 2. Steps for abstracting the effect of iterative statements

Procedure Calls. A procedure *declaration* has the form

$$\text{proc } p \text{ (value } \bar{x}; \text{ value-result } \bar{y}; \text{ result } \bar{z}); \\ \{P\} \langle \text{body} \rangle \{Q\}$$

where \bar{x} represents all the **value** parameters, \bar{y} represents all the **value-result** parameters, and \bar{z} represents all the **result** parameters for the procedure. $\langle \text{body} \rangle$ is one or more statements making up the “procedure”, while $\{P\}$ and $\{Q\}$ are the precondition and postcondition, respectively. The syntactic *signature* of a procedure appears as

$$\text{proc } p : (\text{input_type_name})^* \rightarrow (\text{output_type_name})^*$$

where the Kleene star (*) indicates zero or more repetitions of the preceding unit, *input_type_name* denotes the name of an input parameter to the procedure *p*, and *output_type_name* denotes the name of an output parameter to procedure *p*. A specification of a procedure can be constructed of the form

$$\begin{aligned} & \{ \mathbf{pre}: U \} \\ & \{ \mathbf{post}: \mathit{post}(\mathit{body}) \wedge U \} \\ & \mathbf{proc} \langle \mathit{identifier} \rangle : E_0 \rightarrow E_1 \end{aligned}$$

where E_0 is one or more input parameter types with attribute **value** or **value-result**, and E_1 is one or more output parameter types with attribute **value-result** or **result**. The postcondition for the body of the procedure, $\mathit{post}(\mathit{body})$, is constructed using the previously defined guidelines for assignments, alternatives, and iteratives as applied to the statements of the procedure body.

Gries defines a theorem for specifying the effects of a procedure call [10]. Given a procedure declaration of the above form, the following condition holds

$$\{ PR : P_{\bar{a}, \bar{b}}^{\bar{x}, \bar{y}} \wedge (\forall \bar{u}, \bar{v} : Q_{\bar{u}, \bar{v}}^{\bar{y}, \bar{z}} \Rightarrow R_{\bar{u}, \bar{v}}^{\bar{b}, \bar{c}}) \} p(\bar{a}, \bar{b}, \bar{c}) \{ R \}$$

for a procedure call $p(\bar{a}, \bar{b}, \bar{c})$, where \bar{a} , \bar{b} , and \bar{c} represent the actual parameters of type **value**, **value-result**, and **result**, respectively. PR must hold before the execution of procedure p in order to satisfy R . PR states that the precondition to procedure p must hold for the parameters passed to the procedure and that the postcondition for procedure p implies R for each **value-result** and **result** parameter. Using this theorem for the procedure call, an abstraction of the effects of a procedure call can be derived using a specification of the procedure declaration.

2.3 Specification Language

Predicate logic is the target language for the abstraction algorithms described in this section because its syntax and semantics are well-defined [15]. In addition, it provides a good basis for migration to some other specification language, if desired. Figure 3 gives the format of the specification of procedures where, again, the Kleene star (*) denotes zero or more repetitions of the previous unit, **in** identifies the formal input parameters, **out** identifies the formal output parameters, **local** identifies the local variables of the procedure, $\{\mathbf{pre}: \mathit{expression}\}$ is the precondition of the procedure, and $\{\mathbf{post}: \mathit{expression}\}$ is the postcondition of the procedure.

2.4 Example

Consider the following code sequence written in the Pascal language [1]:

```

proc function_name : (type_name)* → type_name*
in( (variable: type_name)* )
out( (variable: type_name)* )
local ( (variable: type_name)* )
{ pre: expression }
{ post: expression }

```

Fig. 3. Function specification format

```

St = record
    t : integer;
    e : array[1..maxlength] of elementtype
end;

...

function mts ( S : St ) : boolean;
begin
    if S.t > maxlength then
        mts := true
    else
        mts := false
    end;

```

The alternative statement contained within the **begin-end** block can be translated into the following logical expression

$$(((S.t > maxlength) \wedge (mts = true)) \vee (\neg(S.t > maxlength) \wedge (mts = false))) \wedge U,$$

where U represents the precondition for function **mts**. A formal specification for the **mts** function is given in Figure 5, where $domain(S)$ is a predicate that describes the set of all states in which S is defined. The specification of **mts** identifies a single input and a single output based on the function header. The *precondition* asserts that the **value** and **value-result** parameters of the procedure are well-defined with respect to their domain [10]. The *postcondition* is constructed using the abstraction algorithms, where U represents the conditions prior to the current segment of code being processed. The abstracted formal specifications for each procedure contained in Figure 4 were derived using the previously defined guidelines and can be found in Appendix A.

3 Identification of Classes Using Formal Specifications

The formal specifications constructed in Phase 1 have properties that are amenable to the semi-automated identification of objects. An object is a self-contained module

```

program QeS(output);
const
  maxlength = 50;
type
  St = record
    t : integer;
    e : array[1..maxlength] of elementtype
  end;
  Qu = record
    e : array[1..maxlength] of elementtype;
    f, r : integer
  end;
  elementtype : Qu;

procedure mns ( var S : St );
begin
  S.t := maxlength + 1;
end;

function mts ( S : St ) : boolean;
begin
  if S.t > maxlength then
    mts := true
  else
    mts := false
  end;
end;

function tp ( var S : St ) : elementtype;
begin
  if mts(S) then
    writeln('error');
  else
    tp := S.e[S.top]
  end;
end;

procedure po ( var S : St );
begin
  if mts(S) then
    writeln('error');
  else
    S.t := S.t + 1;
  end;
end;

procedure pu ( x : elementtype; var S : St );
begin
  if S.t = 1 then
    writeln('error');
  else
    begin
      S.t := S.t - 1;
      S.e[S.t] := x
    end;
  end;
end;

procedure mnq ( var Q : Qu );
begin
  Q.f := maxlength;
  Q.r := maxlength;
end;

function mtq ( var Q : Qu ) : boolean;
begin
  if Q.r = Q.f then
    mtq := true
  else
    mtq := false
  end;
end;

function fr ( var Q : Qu ) : elementtype;
begin
  if mtq(Q) then
    writeln('error')
  else
    fr := Q.e[Q.f];
  end;
end;

procedure en ( x : elementtype; var Q : Qu );
begin
  if Q.r = maxlength then
    Q.r := 1
  else
    Q.r := Q.r + 1;
  if Q.r = Q.f then
    writeln('error')
  else
    Q.e[Q.r] := x;
  end;
end;

procedure de ( var Q : Qu );
begin
  if mtq(Q) then
    writeln('error')
  else
    if Q.f = maxlength then
      Q.f := 1
    else
      Q.f := Q.f + 1;
    end;
  end;
end;

var
  ex_s : St;
  ex_q : Qu;

begin
  (* QeS Body *)
end

```

Fig. 4. Example Pascal Code

```

proc mts : St → boolean
in( S : St )
out( mts : boolean )
{ pre: domain(S) }
{ post: (((S.t > maxlength) ∧ (mts = true)) ∨
        (¬(S.t > maxlength) ∧ (mts = false))) ∧ domain(S) }

```

Fig. 5. Formal Specification of mts

that includes both the data and procedures that operate on that data. An object can be considered to be an abstract data type (ADT). A class is a collection of objects that have common use [21].

3.1 Guidelines

Using the above definition of an object, a set of guidelines for identifying objects is as follows:

1. Construct a list of all data structures contained within the system. These data structures should not include primitive types.
2. For each data structure contained in the list of system data structures, group together the operations that refer to the data structure as input in the syntactic signature specification of the procedure with the data structure.
3. In the case of conflicts, (i.e., a procedure contains two non-primitive data structures as input in the signature) one of three actions can be taken
 - (a) Determine if one data structure is composed of one or more occurrences of the other data structure. This step is performed by checking the definitions of data structures.
 - (b) Determine whether the output of the procedure excludes either data structure. If it can be shown that the procedure does not modify a data structure then associate the procedure with the data structure that is modified.
 - (c) In the cases where no determination can be made as to how to associate a procedure with a respective data structure, query the domain expert on the appropriate association.

The process of identifying candidate objects is facilitated by the format of the formal specifications for procedures. It is emphasized that the objects identified by this technique are only *candidate* objects. However, once the object definition has been constructed, the formal nature of the specification facilitates formal reasoning about the objects and can aid in the verification of candidate objects as true objects.

3.2 Specification Language

An object specification is constructed by collecting the procedure specifications associated to a data structure and declaring the data structure to be a candidate object. The BNF grammar for the language used to specify potential object classes is given in Figure 6. The definition uses the roman font to describe non-terminals; bold type defines keywords; the Kleene star (*) is used to denote one or more repetitions of the preceding unit; square brackets ([]) indicate optional items; parentheses (‘()’) indicate groupings. The non-terminal, *expression*, represents a predicate logic expression.

```

component = type type_name: has ( method )*
has = has ( data_description )*
data_description =
    variable : type_name
method = method method_name: ( type_name )* → ( type_name )*
        in((variable: type_name)* )
        local((variable: type_name)* )
        out((variable: type_name)* )
        { pre: expression }
        { post: expression }
expression = true
    | false
    | ( expression )
    | ¬ expression
    | expression ∧ expression
    | expression ∨ expression
    | expression ⇒ expression
    | expression ⇔ expression
    | ( ∀ variable : type :: expression )
    | ( ∃ variable : type :: expression )
    | predicate_name [( term ( , term )*)]
    | term def = expression
term = variable
    | function_name [( term ( , term )*)]

```

Fig. 6. Grammar for object specifications

3.3 Example Identification of Candidate Objects

The first step of the analysis is to identify the non-primitive data structures of the system. This process is performed by examining the signatures of the procedure specifications and extracting the unique non-primitive data structure names. For example, analysis of the formal specifications in Appendix A for the program QeS (shown in Figure 4) identifies non-primitive data structures named **St**, **Qu**, and **elementtype**. In continuing the analysis of the data structure **St**, the procedures **mns**, **mts**, **tp**, and **po** are grouped with **St** through examination of the specification

signatures. Procedure `pu` is initially grouped with `St` and `elementtype`, but further analysis leads to a strict association of `pu` to `St` since the input `elementtype` is not modified by `pu`. The subsequent object definition for `St` appears in Figure 7.

```

type St:
has
  t : integer;
  e : array of elementtype;

method mns : St → St
  in( S : St )
  out( S : St )
  { pre: true }
  { post: S.t = maxlen + 1 ∧ true }
method mts : St → boolean
  in( S : St )
  out( mts : boolean )
  { pre: domain(S) }
  { post: (((S.t > maxlen) ∧ (mts = true)) ∨
    (¬(S.t > maxlen) ∧ (mts = false))) ∧ domain(S) }
method tp : St → St × elementtype
  in( S : St )
  out( S : St, tp : elementtype )
  { pre: domain(S) }
  { post: (((S.t > maxlen) ∧ (mts = true)) ∧ post(writeln('error')) ∨
    (¬(S.t > maxlen) ∧ (mts = false)) ∧ (tp = S.e[S.t])) ∧ domain(S) }
method po : St → St
  in( S : St )
  out( S : St )
  { pre: domain(S) }
  { post: (((S.t > maxlen) ∧ (mts = true)) ∧ post(writeln('error')) ∨
    (¬(S.t > maxlen) ∧ (mts = false)) ∧ (S.t1 = S.t0 + 1)) ∧ domain(S) }
method pu : St × elementtype → St
  in( S : St, x : elementtype )
  out( S : St )
  { pre: domain(S) ∧ domain(x) }
  { post: (S.t = 1 ∧ post(writeln('error')) ∨
    (¬(S.t = 1) ∧ (S.t1 = S.t0 ∧ S.e[S.t1] = x)) ∧ domain(S) }

```

Fig. 7. Specification of Object *St*

4 Conclusions and Future Investigations

A requirement to achieving the same functionality of existing software on new hardware platforms and in new programming paradigms has prompted a need for reverse engineering techniques that take advantage of rigorous mathematical methods. Automating the process for abstracting formal specifications from program code is sought but, unfortunately, not completely realizable as of yet. Providing maintenance engineers with as much information as possible, whether it is formal or informal, can aid in the successful construction of specifications that accurately describe legacy systems and thus, facilitate its understanding and re-implementation.

As mentioned earlier, our previous investigations led to the development of AUTOSPEC, a system that abstracts formal specifications from program code using a

translational approach to recognizing the effect of basic programming constructs [4]. Future investigations include extending AUTOSPEC to fully support the two phase approach described in this paper. Additional investigations will address the processing and refinement of candidate objects in order to determine true object definitions. In addition, our investigations into the construction of software component libraries [5, 13] will be used to assist in the identification of object instantiations and class hierarchies. Finally, graphical support for Phase 2 will be used to represent message passing between objects identified in systems.

A Specifications of Example

```

proc mns : St → St
in( S : St )
out( S : St )
{ pre: true }
{ post: S.t = maxlengh + 1 ∧ true }

proc tp : St → St × elementtype
in( S : St )
out( S : St, tp : elementtype )
{ pre: domain(S) }
{ post: (((S.t > maxlengh) ∧ (mts = true)) ∧
  post(writeln('error')) ∨
  ((¬(S.t > maxlengh) ∧ (mts = false)) ∧
  (tp = S.e[S.t])) ∧ domain(S) }

proc po : St → St
in( S : St )
out( S : St )
{ pre: domain(S) }
{ post: (((S.t > maxlengh) ∧ (mts = true)) ∧
  post(writeln('error')) ∨
  ((¬(S.t > maxlengh) ∧ (mts = false)) ∧
  (S.t1 = S.t0 + 1)) ∧
  domain(S) }

proc pu : St × elementtype → St
in( S : St, x : elementtype )
out( S : St )
{ pre: domain(S) ∧ domain(x) }
{ post: (S.t = 1 ∧ post(writeln('error'))) ∨
  (¬(S.t = 1) ∧ (S.t1 = S.t0 ∧ S.e[S.t1 = x])
  ∧ domain(S) ∧ domain(x) }

proc mnq : Qu → Qu
in( Q : Qu )
out( Q : Qu )
{ pre: true }
{ post: Q.f = maxlengh ∧
  Q.r = maxlengh ∧ true }

proc mtq : Qu → Qu × boolean
in( Q : Qu )
out( Q : Qu, mtq : boolean )
{ pre: domain(Q) }
{ post: (Q.r = Q.f ∧ mtq = true) ∨
  (¬(Q.r = Q.f) ∧ mtq = false)
  ∧ domain(Q) }

proc fr : Qu → Qu × elementtype
in( Q : Qu )
out( Q : Qu, fr : elementtype )
{ pre: domain(Q) }
{ post: ((Q.r = Q.f ∧ mtq = true) ∧
  (post(writeln('error')))) ∨
  ((Q.r = Q.f ∧ mtq = true) ∧
  (fr = Q.e[Q.f])) ∧ domain(Q) }

proc en : elementtype × Qu → Qu
in( x : elementtype, Q : Qu )
out( Q : Qu )
{ pre: domain(Q) ∧ domain(x) }
{ post: ((Q.r0 = maxlengh ∧ Q.r1 = 1) ∨
  (¬(Q.r0 = maxlengh) ∧
  (Q.r1 = Q.r0 + 1))) ∧
  (((Q.r = Q.f) ∧ post(writeln('error')))) ∨
  (¬(Q.r = Q.f) ∧ (Q.e[Q.r] = x))
  ∧ domain(Q) }

proc de : Qu → Qu
in( Q : Qu )
out( Q : Qu )
{ pre: domain(Q) }
{ post: ((Q.r = Q.f ∧ mtq = true) ∧
  (post(writeln('error')))) ∨
  ((¬(Q.r = Q.f) ∧ mtq = false) ∧
  (((Q.f = maxlengh) ∧ Q.f = 1) ∨
  (¬(Q.f0 = maxlengh) ∧
  (Q.f1 = Q.f0 + 1)))) ∧ domain(Q) }

```

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algo-*

- rithms*. Addison-Wesley, 1983.
2. Eric J. Byrne and David A. Gustafson. A Software Re-engineering Process Model. In *COMPSAC*. ACM, 1992.
 3. Betty H.C. Cheng. Synthesis of Procedural Abstractions from Formal Specifications. In *Proceedings of COMPSAC'91: Computer Software and Applications Conference*, September 1991.
 4. Betty H.C. Cheng and Gerald C. Gannod. Abstraction of Formal Specifications from Program Code. In *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*. IEEE, 1991.
 5. Betty H.C. Cheng and Jun jang Jeng. Formal methods applied to reuse. In *Proceedings of the Fifth Workshop in Software Reuse*, 1992.
 6. Betty Hsiao-Chih Cheng. *Synthesis of Procedural and Data Abstractions*. PhD thesis, University of Illinois at Urbana-Champaign, 1304 West Springfield, Urbana, Illinois 61801, August 1990. Tech Report UIUCDCS-R-90-1631.
 7. Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, January 1990.
 8. Robert G. Fichman and Chris F. Kemmerer. Object-Oriented and Conventional Analysis and Design Methodologies : Comparison and Critique. *IEEE Computer*, October 1992.
 9. Susan L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, pages 7–10, September 1990.
 10. David Gries. *The Science of Programming*. Springer-Verlag, 1981.
 11. Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.
 12. H.P. Haughton and K. Lano. Objects Revisited. In *Conference on Software Maintenance*. IEEE, 1991.
 13. Jun jang Jeng and Betty H.C. Cheng. Using Automated Reasoning to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, December 1992.
 14. Richard A. Kemmerer. Integrating Formal Methods into the Development Process. *IEEE Software*, pages 37–50, September 1990.
 15. Robert Kowalski. Predicate logic as a programming language. *Information Processing '74, Proceedings of the IFIP Congress*, pages 569–574, 1974.
 16. K. Lano and P.T. Breuer. From Programs to Z Specifications. In *Z User Workshop*. Springer-Verlag, 1989.
 17. Nancy G. Leveson. Formal Methods in Software Engineering. *IEEE Transactions on Software Engineering*, 16(9):929–930, September 1990.
 18. Syng-Syang Liu and Norman Wilde. Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery. In *Conference on Software Maintenance*. IEEE, 1990.
 19. Wilma M. Osborne and Elliot J. Chikofsky. Fitting pieces to the maintenance puzzle. *IEEE Software*, January 1990.
 20. M. Ward, F.W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings Conference on Software Maintenance*, pages 307–315, Miami, Florida, October 1989. IEEE.
 21. A. Winblad, S. Edwards, and D. King. *Object-Oriented Software*. Addison-Wesley, 1990.
 22. Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, September 1990.