

An Automated Approach for Supporting Software Reuse via Reverse Engineering *

Gerald C. Gannod[†]
Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: ggannod@asu.edu

Yonghao Chen and Betty H. C. Cheng
Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824-1226
E-mail: {chenyong, chengb}@cse.msu.edu

Abstract

Formal approaches to software reuse rely heavily upon specification matching criterion, where a search query using formal specifications is used to search a library of components indexed by specifications. In previous investigations, we addressed the use of formal methods and component libraries to support software reuse and construction of software based on component specifications. A difficulty for all formal approaches to software reuse is the creation of the formal indices. We have developed an approach to reverse engineering that is based on the use of formal methods to derive formal specifications of existing programs. In this paper, we present an approach for combining software reverse engineering and software reuse to support populating specification libraries for the purposes of software reuse. In addition, we discuss the results of our initial investigations into the use of tools to support an entire process of populating and using a specification library to construct a software application.

1 Introduction

Historically, the use of software components-off-the-shelf (COTS) has been limited to complete applications. The introduction of object-oriented programming, design patterns, and other new development techniques have focused on the creation and reuse of finer-grained units such

as software COTS, but the wide-scale use of such components in the same manner as hardware integrated components has been limited. As a development technique, *software reuse* is a process of constructing a software system using existing software components. Formal approaches to software reuse rely heavily upon specification matching criterion, where a search query using formal specifications is used to search a library of components indexed by formal specifications. In previous investigations, we addressed the use of formal methods and component libraries to support software reuse [14], and construction of software based on architectural specifications [2, 3]. A difficulty for all formal approaches to software reuse is the creation of the formal indices.

Software reverse engineering is a process of examining system components and component interrelationships in order to derive a description of the system at a higher level of abstraction [4]. We have developed an approach to reverse engineering that is based on the use of formal methods to derive formal specifications of existing programs [8, 9].

In this paper, we present an approach for combining software reverse engineering and software reuse to support populating specification libraries for the purposes of software reuse. In addition, we discuss the results of our preliminary investigations into the use of tools to support an entire process of populating and using a specification library to construct a software application. The remainder of this paper is organized as follows. Section 2 presents background information in the areas of software maintenance, formal methods, and software reuse. A framework for combining the use of software reverse engineering and software reuse is discussed in Section 3. Sections 4 and 5 describe the reverse engineering and reuse engineering aspects of the framework, respectively. An example that illustrates use of the entire framework is described in Section 6. Related work is presented in Section 7. Finally, Section 8 draws conclusions and suggests future investigations.

*This work was supported in part by the NASA Graduate Student Researcher Fellowship NGT-70376, DARPA grant F30602-96-1-0298 (administered by Air Force's Rome Laboratory), and the National Science Foundation grants CCR-9633391, CCR-9407318, CCR-9209873, and CDA-9312389.

[†]This author was supported in part by a NASA Graduate Student Researchers Program Fellowship. This research was performed while this author was a Ph.D. student at Michigan State University. A portion of this research was performed while this author was at the NASA Jet Propulsion Laboratory.

2 Background

This section gives background information in the areas of software maintenance, software reuse and formal methods for software development.

2.1 Software Maintenance

Figure 1 contains a graphical depiction of a process model for reverse engineering and reengineering [1]. The process model is captured by two sectioned triangles, where each section in a triangle represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. Entry into this reengineering process model begins with system *A*, where *Abstraction* (or reverse engineering) is performed to a level of detail appropriate to the task being performed. For instance, if a system is to be reengineered in response to efficiency constraints, then abstraction to the design level may be appropriate. The next step is *Alteration*, where the system is configured into a new form at a different level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system *B*. The context of the investigations described in this paper corresponds to the implementation and design levels of abstraction.

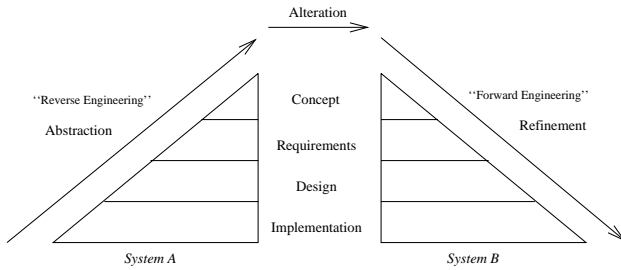


Figure 1. Reverse Engineering Process Model

2.2 Formal Methods

Although the waterfall development life-cycle provides a structured process for developing software, the design methodologies that support the life-cycle (i.e., Structured Analysis and Structured Design [24]) make use of informal techniques, thus increasing the potential for introducing ambiguity, inconsistency, and incompleteness in designs and implementations. In contrast, formal methods used in software development are rigorous techniques for specifying, developing, and verifying computer software [22]. A formal method consists of a well-defined specification language with a set of well-defined inference rules that can be used to reason about a specification [22]. A benefit of formal methods is that their notations are well-defined and thus, are amenable to automated processing.

2.3 Strongest Postcondition

In previous investigations, we described the use of strongest postcondition as the formal basis for reverse engineering [8], and used strongest postcondition to analyze NASA JPL flight software written in C [9]. *Strongest postcondition*, denoted $sp(S, Q)$ is defined as follows: given that Q holds, execution of S results in $sp(S, Q)$ true, if S terminates [5]. As such, $sp(S, Q)$ assumes partial correctness. In essence, given a precondition Q and a program S , the strongest postcondition sp derives a predicate $sp(S, Q)$.

2.4 Software Reuse

Software reuse is the use of existing software components to construct new systems [15]. Jeng and Cheng developed a formal approach for specifying and classifying components in a reusable library [12, 13]. They also introduced the use of a generality operator as the formal basis for identifying reusable components via specification matching [12]. Zaremski and Wing [25] describe additional operators for matching queries to components for software reuse. In addition, Penix and Alexander [19] define the *satisfies* criterion for component matching. Figure 2 shows several of the matching operators and their relations. When these operators are used for software reuse, A is a query specification and R is a library specification.

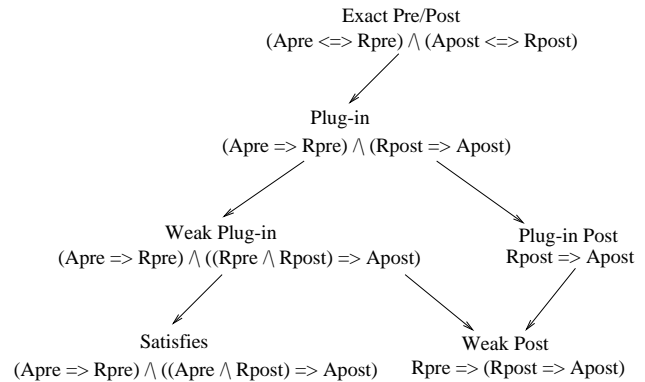


Figure 2. A partial-order of matching operators

In Figure 2, an arrow represents a logical implication between two matching operators. When used to select reusable components, all the matching operators along the leftmost branch in the figure ensure that a selected component can be reused for implementing a query specification without semantic adaptation.¹ This semantic requirement is relaxed in the cases of “plug-in post” and “weak

¹Syntax mismatches may still exist, thus syntactic adaptation may be needed.

post” matches, where logical implications are not required between the preconditions of the two matched functions. Adaptation may be needed to establish the precondition of the reused component in both plug-in post and weak post matches. Section 4 discusses how these matching relationships can be used to facilitate the design abstraction process for reverse engineering purposes.

3 A Software Reverse Engineering and Reuse Framework

Many software reuse approaches assume that a library of reusable components exists. There are two techniques for populating these libraries with reusable code. The first technique is to construct components with the intention of placing them into code repositories. Examples of these repositories are the Microsoft Foundation Classes [16] as well as libraries for standard problem domains such as mathematics, graphics, and networking. The second technique for populating code repositories is to identify existing code as potential candidates for reuse, and then packaging that code into a library. In either case, a primary concern is the mechanisms used for indexing, identifying, and retrieving the components from the libraries [6, 14, 19, 25].

This paper describes how a formal approach to reverse engineering can be integrated with a formal approach to software reuse in order to support after-the-fact construction and use of reusable code libraries. The overall process that is presented consists of four steps. First, a reuse library is populated by identifying reusable code and reverse engineering that code in order to extract indices for the components. The components are then organized according to logical relationships that exist between code entities. Second, a specification of a target system is used to search the reuse library for components that satisfy desired behavior. Third, candidate components are retrieved and a match is established. Fourth, existing components are adapted (as necessary) and packaged into the target system.

Figure 3 gives an overview of the reverse engineering and software reuse framework in the form of a data-flow diagram. The diagram shows two distinct components within the framework: the *Reverse Engineering* component, indicated by the process circle labeled “RevEgr Suite”, and the *Reuse* component, indicated by the process circle labeled “Reuse Suite”. Two integrating factors within this framework provide the linkage between the two components; the *User* and the *Specification Library*. The user is required to direct the reverse engineering and the reuse processes, and the specification library is the common medium and repository between the RevEgr Suite and the Reuse Suite.

Within this framework, a user can analyze source code and construct formal specifications that can be used to index the source for reuse purposes. This reverse engineering and population activity facilitates the reuse part of the frame-

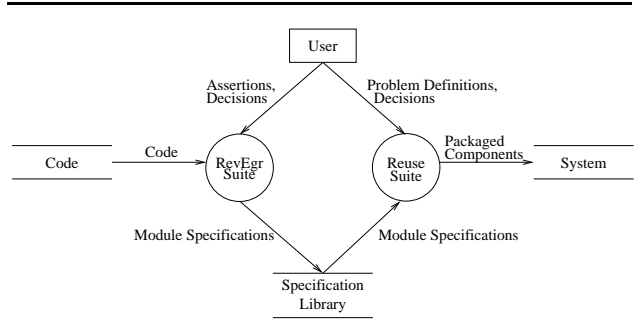


Figure 3. The Reverse Engineering and Reuse Framework

work, namely the search, identification, and packaging of components into new systems. Sections 4 and 5 discuss the specific techniques and tools that are used to support the reverse engineering and reuse aspects of the framework, respectively.

4 Reverse Engineering

In this section we describe an approach for reverse engineering that is applicable to the implementation and design levels of abstraction shown in Figure 1.

4.1 Approach

In previous work, we investigated two phases of software reverse engineering and design recovery. The first phase involves the construction of low-level, *as-built* specifications from program code [8]. The second phase of our investigations into reverse engineering involves the derivation of high-level abstractions from as-built specifications [10]. By constructing high-level abstractions, several activities can be facilitated including high-level reasoning, program understanding, and software reuse.

The combination of these two investigations provides the basis for an overall process for reverse engineering that involves two steps: 1) the construction of as-built specifications from program code, and 2) the derivation of high-level specifications from the as-built specifications.

4.2 As-Built Specifications

As stated in Section 2, the *strongest postcondition* (denoted *sp*) is defined as the strongest condition *R* that is true after the execution of a program *S*, when starting with condition *Q* true. Table 1 summarizes the strongest postcondition semantics of the Dijkstra guarded command language [5], where IF represents the *n* alternative conditional statement

```

if
  B1 → S1;
  ...
  || Bn → Sn;
fi;
  
```

$B_i \rightarrow S_i$ represents a guarded command such that S_i is only executed if logical expression (guard) B_i is true. `DO` represents the loop statement “do $B \rightarrow S$ od” where S is executed iteratively until guard B is false.

Construct	sp Semantics
$sp(x := e, Q)$	$\equiv (\exists v :: Q_v^x \wedge x = e_v^x)$
$sp(\text{IF}, Q)$	$\equiv sp(S_1, B_1 \wedge Q) \vee \dots \vee sp(S_n, B_n \wedge Q)$
$sp(\text{DO}, Q)$	$\equiv \neg B \wedge (\exists i : 0 \leq i : sp(\text{IF}^i, Q))$
$sp(S_1; S_2, Q)$	$\equiv sp(S_2, sp(S_1, Q))$

Table 1. Strongest Postcondition Semantics for the Dijkstra Guarded Command Language

In the table, the semantics for $sp(x := e, Q)$ states that after the execution of “ $x := e$ ” there exists some value v such that every free occurrence of x in Q is replaced with v and $x = e_v^x$. The semantics for $sp(\text{IF}, Q)$ states that after execution of the `if-fi` statement, at least one of $sp(S_i, B_i \wedge Q)$ is true. In the case of iteration, denoted $sp(\text{DO}, Q)$, the semantics are that after execution of the loop, the loop guard is false ($\neg B$), and a disjunctive expression describing the effects of iterating the loop some number of times (approximated by the notation IF^k) is true, where $k \geq 0$. Finally, for sequences, $sp(S_1; S_2, Q)$ means that the postcondition for statement S_1 is the precondition for some subsequent statement S_2 .

4.3 Deriving Abstractions

Section 2 described several specification matching criteria that have been used for software component reuse purposes. For the purposes of reverse engineering and program understanding, we have defined the concept of an *abstraction match* as follows [10]:

Definition 1 (Abstraction Match) *Let \mathcal{I} be an existing program with specification i such that the corresponding precondition and postcondition are i_{pre} and i_{post} , respectively, and let l be an axiomatic specification with precondition l_{pre} and postcondition l_{post} . A match is an **abstraction match** if $i \preceq l$, so that*

$$(l_{pre} \rightarrow i_{pre}) \wedge (i_{post} \rightarrow l_{post}).$$

One of the interesting properties of some of the matching operators in Figure 2 is that they are partial-order relations [10]. As such, high-level abstractions of axiomatic specifications can be derived as long as the matching criterion are preserved. That is, given an axiomatic specification I that consists of precondition I_{pre} and postcondition I_{post} , we would like to identify an axiomatic specification A such that $I \preceq A$. We can identify such a specification by modifying I so that we have a specification I' that sat-

isfies the relationship that $I \preceq I'$. If, for instance, \preceq is the abstraction match operator, then by either strengthening the precondition I_{pre} , weakening the postcondition I_{post} , or both, we produce a specification I' that satisfies the property that $I \preceq I'$. In order to address the computational complexities of deriving abstractions based on matching, we have developed a number of guidelines that focus on weakening the postcondition I_{post} and strengthening the precondition I_{pre} [10].

One technique for preserving an abstraction match relationship is by weakening the postcondition of a specification. Let I be a specification with precondition I_{pre} and postcondition I_{post} and let I' be a specification such that $I'_{pre} \leftrightarrow I_{pre}$ and $I_{post} \rightarrow I'_{post}$. As such, $I \preceq I'$, since

$$\begin{aligned} & ((I'_{pre} \leftrightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})) \\ & \Rightarrow \\ & ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})). \end{aligned} \quad (1)$$

Expression (1) provides a basis for deriving abstractions from a specification by weakening a postcondition I_{post} to produce a postcondition I'_{post} . Several options are available for weakening the postcondition including those listed in Table 2, which includes *delete a conjunct*, *add a disjunct*, \wedge to \vee transformation, and \wedge to \rightarrow transformation.

Operation	I_{post}	I'_{post}
Delete a conjunct	$A \wedge B \wedge C$	$A \wedge C$
Add a disjunct	$A \wedge B$	$(A \wedge B) \vee C$
\wedge to \rightarrow	$A \wedge B$	$A \rightarrow B$
\wedge to \vee	$A \wedge B$	$A \vee B$

Table 2. Weakening the postcondition

In Section 6.1 we describe an example that applies the use of the *delete a conjunct* strategy for deriving abstractions of as-built specifications. The remainder of this section presents *delete a conjunct* strategy and conditions for its application. For further discussion about the remaining strategies, please refer to [10].

Delete a conjunct. Given a specification in conjunctive form (not necessarily a normal form), deletion of a conjunct weakens a specification by removing additional or constraining conditions. Below are guidelines that can be used to identify the appropriate conjunct for deletion.

Local Scope: If a conjunct specifies behavior that is local to a procedure and has no impact on the output variables of the system, then that conjunct is a candidate for deletion. Examples include specifications of the value of a loop index or temporary variables.

Independence: If a conjunct specifies some behavior that is logically independent of the remaining conjuncts,

then that conjunct is a candidate for deletion. As an example, consider the expression $(x = c) \wedge (c = y) \wedge (z = n)$. The conjunct $(z = n)$ is independent of the conjuncts $(x = c)$ and $(c = y)$.

Preservation: If a conjunct captures some behavior that must be expressed in the higher level specification, then the remaining conjuncts are candidates for deletion.

These guidelines are by no means comprehensive. Ultimately, a maintenance engineer using this approach must decide whether to delete a specific conjunct in a specification

4.4 Tool Support

In order to support the techniques described in Sections 4.2 and 4.3, we have been developing a suite of prototype analysis tools that facilitate both phases of the reverse engineering process. The first tool, called AUTOSPEC [8], is used by the maintenance programmer to construct an as-built formal specification from program code. A second tool, called SPECGEN [10], is used by the maintenance programmer to derive formal high-level abstractions from the as-built specifications that are constructed by AUTOSPEC.

Figure 4 shows a data-flow diagram depicting the context of the tools used to support the two phase reverse engineering technique, where circles denote processes, and arrows depict data flow. The AUTOSPEC tool takes source code and user-defined assertions as input, and with user guidance, produces as-built specifications that can be used as input to the SPECGEN tool. SPECGEN uses the abstraction matching technique to then derive high-level module specifications based on user decisions to populate a library of specifications.

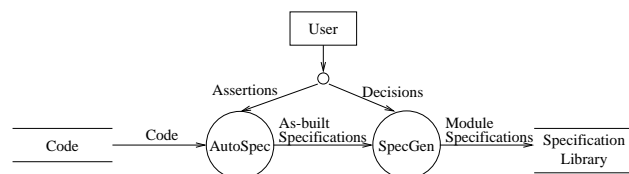


Figure 4. Reverse Engineering Component

The convention used in this paper for the as-built specifications generated by AUTOSPEC is given in Figure 5. The format is based on the Larch interface language [11] syntax. In this convention, *domainsort* and *rangesort* are the input and output types of a given function, respectively. The **locals** keyword lists the variables defined within the scope of the specification, if applicable. The **requires** keyword is used to indicate the precondition of the given function. The **ensures** keyword describes the postcondition of a given function. Finally, the **modifies** keyword lists the variables that are modified by the function.

```
spec name ( (var: domainsort)* ) → var: rangesort
  locals (var: domainsort)*
  requires precondition
  modifies variables
  ensures postcondition
```

Figure 5. Syntax of Library Specifications

5 Software Reuse

Although a large number of techniques have been developed to address various reuse issues, the lack of a seamless integration of these techniques imposes significant obstacles to achieving effective reuse. Previously, we developed an architecture-based framework, called ABRIE (Architecture Based Reuse and Integration Environment) for reuse-based software development [2]. Within this framework, reuse issues are addressed in an integrated and automated fashion. Figure 6 depicts the ABRIE framework in the form of a data flow diagram, where the dashed line encapsulates the ABRIE system.

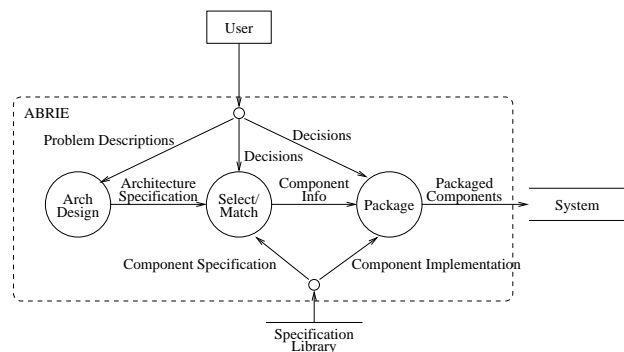


Figure 6. Software Reuse Framework

As shown in Figure 6, given the problem definition, software development starts with architecture design. By specifying the constituent components and their interrelationships for a target system, the software architecture specification serves as a basis for evaluating and integrating existing components. The evaluation and selection of existing components are based on both architectural characteristics and behavioral specifications. Mismatches may be identified in the component evaluation and matching process. Based on the selected components and matching information, the packaging process generates a system construction file that describes how the target system can be constructed. Identified mismatches may also be resolved in the packaging process by generating appropriate wrappers for the reused components.

6 Example

In this section we discuss an example that illustrates the use of the integrated reverse engineering and reuse framework. First, we populate a library, using reverse engineering techniques, with component specifications. Then we demonstrate the process of specifying an application and searching the specification and component library for suitable reusable code. Finally, we show the process of packaging components into the final application.

6.1 Populating the Library

Figure 7 shows the source code for the `enQueue` procedure for an array implementation of a queue abstract data type. The source code, written using the C programming language, implements a circular queue so that the head and tail of the queue can “wrap” around the lowest and highest indices of the array, as shown in Figure 8.

```
int enQueue(Queue *q, QDATA *e) {
    int tail;
    int head;

    if ((q->tail - q->head) == MAXSIZE)
    {
        printf("Full\n");
        return 0;
    } else {
        q->data[q->tail % MAXSIZE] = *e;
        q->tail = q->tail + 1;
        return 1;
    }
}
```

Figure 7. `enQueue` Source Code

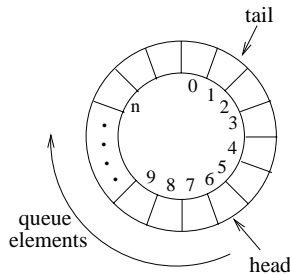


Figure 8. Circular Queue Diagram

The queue data structure consists of three parts: an index to the front, or *head* of the queue, an index to the end, or *tail* of the queue, and an array that is used to store the elements of the queue. The queue implementation contains several functions that correspond to the operations typically associated with queues including `enQueue`, `dequeue`, `new_queue`, `head`, and `is_empty`. The abstract behavior of these operations is as expected, where `enQueue` adds a new ele-

ment to the end of the queue, `dequeue` removes the element at the front of the queue, `new_queue` creates a new queue, `head` returns the element at the front of the queue, and `is_empty` checks to see if the queue contains any elements.

As described in Section 4.1, construction of a specification that is suitable for populating a component specification library involves two primary steps: 1) construction of an as-built specification, and 2) derivation of a high-level abstraction. Figure 9 shows the as-built specification of the `enQueue` procedure as derived by the AUTOSPEC tool. Informally, the as-built specification of the `enQueue` procedure states that prior to the execution of the procedure, the pointer `e` points to an object `_param4`, the value of `_param4` is `_pVal5`, and the pointer `q` points to an object `_param3` such that the `tail` component of `_param3` has the value `_pVal4`. In addition, the specification states that after execution of the procedure, one of two conditions is true. The first condition describes the behavior when the queue is full in which case the difference between the values `_param3.tail.V` and `_param3.head.V` is equal to the maximum size of the queue. Here, the return value of the procedure is 0, indicating a failure. The second condition describes the behavior when there is room to place an item on the queue. In this case, the return value of the procedure is 1, the index to the tail is incremented, and the data element is added to the queue data array.

```
spec int enQueue(Queue *q, QDATA *e)
requires
  ((e .> _param4) &&
   (_param4.V == _pVal5)) &&
  ((q .> _param3) &&
   (_param3.tail.V == _pVal4))
modifies
  q (_param3)
ensures
  (((((e .> _param4) && (_param4.V == _pVal5)) &&
    ((q .> _param3) && (_param3.tail.V == _pVal4))) &&
   ((param3.tail.V - param3.head.V) == MAXSIZE)) &&
   (return.V = 0)) ||
  ((((((e .> _param4) && (_param4.V == _pVal5)) &&
    (q .> _param3) && (_param3.tail.V == _pVal4))) &&
   (!(pVal4 - param3.head.V) == MAXSIZE))) &&
   (_param3.data[(pVal4 % MAXSIZE)].V = _param4.V)) &&
   (_param3.tail.V = (pVal4 + 1))) &&
   (return.V = 1)))
```

Figure 9. Output generated by AUTOSPEC for the `enQueue` procedure

While the specification in Figure 9 is accurate with respect to the original source program, the level of detail can inhibit high-level reasoning. As described in Section 4.3, one technique that can be used to derive an abstraction of an as-built specification is to weaken the postcondition by using the *delete a conjunct* strategy. For the `enQueue` example, we must first put the **ensures** clause into a conjunctive form, as shown in Figure 10. In this form, the `enQueue`

specification has four conjuncts that specify that (a) e points to the object $_param4$, (b) $_param4.V$ has value $_pVal5$, (c) q points to $_param3$, and (d) the disjunctive statement:

```

(((\_param3.tail.V = \_pVal4) &
  ((\_param3.tail.V - \_param3.head.V) = MAXSIZE)) &
  (return.V = 0)) ||
(((as_const9 = \_pVal4) &
  (!((\_pVal4 - \_param3.head.V) = MAXSIZE))) &
  (\_param3.data[as_const9 % MAXSIZE].V = \_param4.V)) &
  (\_param3.tail.V = (as_const9 + 1))) &
  (return.V = 1)))

(e .> \_param4) && (\_param4.V = \_pVal5) && (q .> \_param3)
&& (((\_param3.tail.V = \_pVal4) &
  ((\_param3.tail.V - \_param3.head.V) = MAXSIZE)) &
  (return.V = 0)) ||
((((as_const9 = \_pVal4) &
  (!((\_pVal4 - \_param3.head.V) = MAXSIZE))) &
  (\_param3.data[as_const9 % MAXSIZE].V = \_param4.V)) &
  (\_param3.tail.V = (as_const9 + 1))) &
  (return.V = 1)))

```

Figure 10. The enqueue ensures clause in a conjunctive form

Figure 11 shows the results generated by the SPECGEN tool when applied to the specification in Figure 10. One of the operations that can be performed by the tool is to generate all the possible abstractions of a specification based on preserving one or more of the conjuncts in the specification. The representation of the specifications that are generated by preserving conjuncts is called a *focus graph*. In our example, we are interested in preserving the conjuncts (b) and (d) and deleting conjuncts (a) and (c) due to the independence property stated in Section 4.3. The vertex labeled “abcd” indicates that conjuncts (a), (b), (c), and (d) are conjoined. This vertex corresponds to the original specification in Figure 10. The remaining vertices in the graph represent the possible abstractions that are formed by deleting conjunct (a), (c), or both.

Using the information provided by SPECGEN, several transformations of the specification in Figure 10 can be performed that simplify and introduce abstraction into the postcondition. First, based on the focus graph in Figure 11, conjuncts (a) and (c) are deleted due to the independence criteria. After the deletion of conjuncts (a) and (c), we can perform a textual substitution of all references to the $_param3$ identifier with a more descriptive symbol, like Q . Finally, given that the term as_const9 has the value $_pVal4$ and in the precondition for the specification $_param3.tail.V == _pVal4$, we can replace as_const9 with the term $Q.tail^{\wedge}$, which represents the pre-value for the tail component of the queue (i.e., the value of the tail component before execution of the procedure). Given these transformations, the abstraction for the enqueue procedure can be derived as shown in Figure 12. Informally, the specification states that after execution of the procedure, the return value is set to 0 when the difference

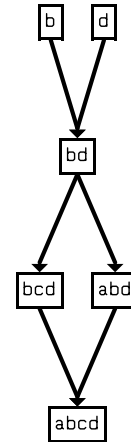


Figure 11. SPECGEN Interface and Output

between the head and tail of the queue is the maximum array size. When the difference between the head and tail is not the maximum array size, then the element E is added to the array at the tail index, the tail index is incremented, and the return value is set to 1.

```

spec int enqueue(Queue *q, QDATA E)
requires
  ((q .> Q) &&
  (Q.tail == Q.tail^))
modifies
  Q.tail, Q.data
ensures
  (((Q.tail - Q.head) = MAXSIZE) &&
  (return = 0)) ||
  (((!((Q.tail^ - Q.head) = MAXSIZE)) &&
  (Q.data[Q.tail^ % MAXSIZE] = E)) &&
  (Q.tail' = (Q.tail^ + 1))) &&
  (return = 1)))

```

Figure 12. The enqueue abstraction

A process similar to the one used for enqueue can be applied to derive abstractions for the remainder of the queue as-built specifications. For the purposes of combining the reverse engineering suite with the reuse suite, the resulting specifications must be translated into the syntax for the ABRIE system. The reason for the differences between the syntax of the AUTOSPEC and ABRIE specification languages is historical [7]. Appendix A contains the module specification in the ABRIE syntax that was constructed for the example described in this section.

6.2 Specifying an Application

In the following discussion, we describe how a solution to the Josephus problem [21] can be specified and assembled from reusable components in ABRIE. In particular, we show

how the formal specifications generated by the reverse engineering process can be used to semantically determine the reusability.

The Josephus game can be described as follows: N people, numbered 1 to N , are sitting in a circle. Starting at person 1, a hot potato is passed. After M passes, the person holding the hot potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining persons wins. Given N and M , the Josephus problem is to determine who will win.

Figure 13 shows the structure of a solution to the Josephus problem that uses a queue to represent people sitting in a circle. The solution is specified in ABRIE. Component *master* simulates the game, and calls queue op-

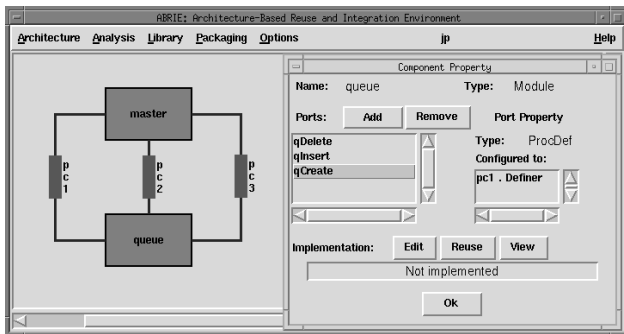


Figure 13. Architecture of a solution to Josephus problem

erations provided by component *queue*. The two components are connected through three connectors of procedure calls. As shown in the “Component Property” window of Figure 13, component *queue* has three ports, each of which defines and provides a queue operation. Figure 14 shows the textual specification of component *queue*. As shown in Figure 14, the required behaviors of its ports have been specified. In the next subsection, we discuss how a library component can be selected based on these behavioral specifications to implement the *queue* interface.

6.3 Component Reuse

ABRIE incorporates a library manager for organizing and managing existing components. Components are classified and retrieved based on their interfaces (i.e., types and ports). When implementing an abstract component (interface) in an architecture, a single click on the *reuse* button in the “Component Property” window (see Figure 13) triggers ABRIE to search the current library (which is loaded through the library manager). All components of the same type as the query interface will be presented to the user. Based on their specifications, the user selects one candidate for further evaluation. Suppose we select *circqueue* for

```

Module queue
  Ports
    ProcDef qCreate() return Queue* {
      uses auxTheories;
      ensures
        result.head=0 /\ result.tail=0;
    }
    ProcDef qInsert(Queue* q,int i) return Bool {
      uses auxTheories;
      modifies q;
      ensures
        (q.tail-q.head = MAXSIZE => result = false)
        /\ (q.tail^ -q.head^ ~= MAXSIZE
            => (result = true
              /\ q.tail' = q.tail^ + 1
              /\ q.data[mod(q.tail^, MAXSIZE)] = i));
    }
    ProcDef qDelete(Queue* q) return int {
      uses auxTheories;
      requires q.head^ ~= q.tail^;
      modifies q;
      ensures
        result=q.data[mod(q.head^, MAXSIZE)]
        /\ q.head'=q.head^+1;
    }
  }
End

```

Figure 14. Specification of component *queue*

implementing *queue*. In order to determine the reusability of *circqueue* (see specification in Appendix A), we need to establish a mapping from the ports of the target component *queue* to those of *circqueue* so that each operation specified in *queue* can be implemented by a corresponding operation in *circqueue*. We conjecture that *qDelete* can be matched (implemented) by *dequeue*, *qInsert* by *enQueue*, and *qCreate* by *new_queue*. Given a match, specification-based proof obligations will be generated to validate the matching. The Larch Prover (LP), an interactive theorem proving system for multi-sorted first-order logic [11], is integrated in ABRIE and can be invoked to assist in proving these obligations. For the above matching process, all the obligations are resolved, thus establishing the reusability of *circqueue* for implementing *queue*. Due to space constraints, screen captures for ABRIE’s matching and validation utilities are not included [7].

As exhibited in the mapping between *circqueue* and *queue*, naming conflicts, such as *qDelete* of *queue* and *dequeue* of *circqueue*, may exist between a query specification and the reused component. Resolving these mismatches is one of the tasks of the packaging process. Figure 15 shows the wrappers generated by ABRIE for resolving the naming conflicts between *circqueue* and *queue*, where wrappers are generated based on the established port mappings. The packaging process also checks connectors and generates their implementation, as well as a system construction file (a *makefile*) that describes how an executable system is produced.

```

// _circqueue_wrapper.cc
// Generated by ABRIE for
// wrapping component circqueue

#include "auxTypes.h"

extern int dequeue(Queue *);
int qDelete(Queue *q) {
    return dequeue(q);
}

extern Bool enqueue(Queue *, int);
Bool qInsert(Queue *q, int i) {
    return enqueue(q, i);
}

extern Queue* new_queue();
Queue *qCreate() {
    return new_queue();
}

```

Figure 15. Wrappers generated by ABRIE for resolving naming conflicts

7 Related Work

Several approaches and experiences have been described for finding reusable components in program code [18, 23]. Neighbors [18] described an approach for constructing reusable components from large existing systems by identifying the tightly coupled subsystems contained in the system. The approach is primarily based on the use of informal and experimental methods. In contrast, the approach described in this paper enables the use of automated reasoning techniques for generating specifications and checking matches between target specifications and library specifications. Yang *et al.* [23] describe a formal approach to identification of reusable components via program understanding. While the goals of their approach are similar to the reverse engineering component described in this paper, the technique is different in that their approach is based on the use of program transformations (a form of rewriting where a library of rules is used to replace one part of a program or specification with another) within the context of a wide spectrum language. In contrast, our approach is based on the use of *program translation*, where an atomic set of rules is used to rigorously derive specifications.

Many approaches have been suggested for the retrieval of components from reusable component libraries, ranging from classification of search criteria [19, 25] to retrieval [6, 14] and library structuring [17]. Jeng and Cheng describe the use of *analogy* and *generality* [12] as the basis for matching functions. Zaremski and Wing have proposed a technique for signature and specification matching [25]. Fischer *et al.* have described an approach for retrieval of reusable components using filters to narrow the search space [6]. While many of these approaches have made significant ad-

vances in the area of software reuse, few address the creation of specification libraries.

8 Conclusions and Future Investigations

Specification libraries have been used extensively and with relative success for code derivation [20]. In addition, formal specification libraries are the basis for many software component reuse approaches [6, 14, 19, 25]. One of the difficulties of using a formal approach to software component reuse is that the assumption of library existence may not be a reasonable one. However, by applying a formal reverse engineering technique to candidate reusable components, the arguments against the existence assumption can be mitigated. As such, an entire framework for constructing systems based on formal methods via software reuse can be facilitated.

One of the benefits of using a formal approach is that the formal notations facilitate the use of tools to automate software development processes, including software reverse engineering and reuse. To this end, the AUTOSPEC, SPEC-GEN, and ABRIE tools have been developed to support the approaches described in this paper. In order to address problems of scale, the AUTOSPEC tool is being combined with a host of other semi-formal support tools to provide an environment for supporting the understanding of C programs. In addition, the SPEC-GEN tool is being updated to facilitate the introduction of several strategies for simplifying as-built formal specifications. Finally, further work is being performed to broaden the reuse strategies supported by ABRIE.

References

- [1] E. J. Byrne. A Conceptual Foundation for Software Re-engineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.
- [2] Y. Chen and B. H. C. Cheng. Facilitating an automated approach to architecture-based software reuse. In *Proceedings of the 12th International Conference on Automated Software Engineering*, 1997.
- [3] Y. Chen and B. H. C. Cheng. Formalizing and automating component reuse. In *Proc. of 9th IEEE Intl. Conference on Tools with Artificial Intelligence*, November 1997.
- [4] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [5] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [6] B. Fischer, M. Kievernagel, and W. Struckmann. Deduction-Based Software Component Retrieval. In *Proceedings of IJ-CAI '95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, August 1995.
- [7] G. C. Gannod, Y. Chen, and B. Cheng. An automated approach for supporting software reuse via reverse engineering. Technical Report MSUCPS:TR98-16, Michigan State University, Department of Computer Science and Engineering, May 1998.

[8] G. C. Gannod and B. H. C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996. A preliminary version appeared in the *Proceedings for the IEEE Second Working Conference on Reverse Engineering*, July 1995.

[9] G. C. Gannod and B. H. C. Cheng. Using Informal and Formal Methods for the Reverse Engineering of C Programs. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 265–274. IEEE, 1996. Also appears in the Proceedings for the Third IEEE Working Conference on Reverse Engineering.

[10] G. C. Gannod and B. H. C. Cheng. A specification matching based approach to reverse engineering. Technical Report MSUCPS-TR98-15, Michigan State University, April 1998.

[11] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[12] J. Jeng and B. H. C. Cheng. Using Automated Reasoning Techniques to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, December 1992.

[13] J. Jeng and B. H. C. Cheng. Using Formal Methods to Construct a Software Component Library. *Proc. of Fourth European Software Engineering Conference*, September 1993. Published in Lecture Notes of Computer Science by Springer-Verlag.

[14] J.-J. Jeng and B. H. Cheng. Specification Matching for Software Reuse: A Foundation. In *Proceedings of the ACM Symposium on Software Reuse*, pages 97–105, 1995.

[15] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), June 1992.

[16] Microsoft Corporation. *Microsoft Visual C++ MFC Library Reference, Part 1 & 2*, 1997.

[17] A. Mili, R. Mili, and R. T. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7), July 1997.

[18] J. M. Neighbors. Finding reusable components in large systems. In *Proceedings of the Third IEEE Working Conference on Reverse Engineering*, pages 2–10. IEEE, November 1996.

[19] J. Penix and P. Alexander. Toward Automated Component Adaptation. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[20] D. R. Smith and E. A. Parra. Transformational Approach to Transportation Scheduling. In *Proceedings of the 18th Knowledge-Based Software Engineering Conference*, pages 60–68, Sept 1993.

[21] M. A. Weiss. *Algorithms, Data Structures, and Problem Solving with C++*. Addison-Wesley Publishing Company, Inc., 1996.

[22] J. M. Wing. A Specifier’s Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[23] H. Yang, P. Luker, and W. C. Chu. Code understanding through program transformation for reusable component identification. In *Proceedings of the Fifth IEEE International Workshop on Program Comprehension*, pages 148–157. IEEE, May 1997.

[24] E. Yourdon and L. Constantine. *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.

[25] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

A Circular Queue Library Specification

Figure 16 shows the library specification for the queue source code. The format for the specifications is based on the Larch interface language [11] and is used by the ABRIE system as a means for storing specifications and references to supporting source code.

```

Module CircularQueue
  Ports
    ProcDef dequeue(Queue* q) return int {
      uses auxTheories;
      requires true;
      modifies q.head;
      ensures
        (q.head^ ~= q.tail^ /\
         q.head' = q.head^ + 1 /\
         result = q.data[mod(q.head^,MAXSIZE)])
        /\
        (q.head' = q.head^ /\
         q.head^ = q.tail^ /\ result = 0);
    }

    ProcDef enqueue(Queue* q, int e) return int {
      uses auxTheories;
      requires true
      modifies q.tail, q.data;
      ensures
        (q.tail - q.head = MAXSIZE) /\ result = 0
        /\
        (q.tail^ - q.head^ ~= MAXSIZE /\
         q.data'[mod(q.tail^,MAXSIZE)] = e /\
         q.tail' = q.tail^ + 1 /\
         result = 1);
    }

    ProcDef head(Queue q) return int {
      uses auxTheories;
      requires true;
      ensures result = q.data[mod(q.head^,MAXSIZE)];
    }

    ProcDef is_empty(Queue q) return Bool {
      uses auxTheories;
      requires true
      ensures result = (q.head == q.tail);
    }

    ProcDef new_queue() return Queue* {
      uses auxTheories;
      requires true;
      ensures
        o.head = 0 /\ o.tail = 0 /\
        result = o;
    }

  Implementation
    source ("/user/r02/chengb/gannod/Research/
           CircQueue/queue/", "queue.c")
End

```

Figure 16. Circular Queue Library Specification
