11-1-2013

Finding Specifications of While Statements Using Patterns

Aditi Barua
University of Texas at El Paso, abarua@miners.utep.edu

Yoonsik Cheon
University of Texas at El Paso, ycheon@utep.edu

Comments:
Technical Report: UTEP-CS-13-67

Finding Specifications of While Statements Using Patterns

Aditi Barua and Yoonsik Cheon

TR #13-67
November 2013

Keywords: code pattern, functional program verification, intended functions, program specification, specification pattern, while statement.

1998 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications—languages; D.2.4 [*Software Engineering*] Software/Program Verification—correctness proofs, formal methods; D.3.3 [*Programming Languages*] Language Constructs and Features—control structures; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—logics of programs, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

Finding Specifications of While Statements Using Patterns

Aditi Barua and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A
abarua@miners.utep.edu; ycheon@utep.edu

***Abstract*—A formal correctness proof of code containing loops such as while statements typically uses the technique of proof-by-induction, and often the most difficult part of carrying out an inductive proof is formulating a correct induction hypothesis, a specification for a loop statement. An incorrect induction hypothesis will surely lead to a proof failure. In this paper we propose a systematic way for identifying specifications of while statements. The key idea of our approach is to categorize and document common patterns of while statements along with their specifications. This is based on our observation that similarly-structured while statements frequently have similarly-structured specifications. Thus, a catalog of code and specification patterns can be used as a good reference for finding and formulating a specification of a while statement. We explain our approach using functional program verification in which a program is viewed as a mathematical function from one program state to another, and a correctness proof is done by comparing two functions, the implemented and the specified. However, we believe our approach is applicable to other verification techniques such as Hoare logic using pre- and post-conditions.**

***Keywords*—code and specification patterns, functional program verification, intended function, specification, while statement.**

I. Introduction

Functional program verification is a formal verification technique originated from the *Cleanroom Software Engineering* [1], in which a program is viewed as a mathematical function from one program state to another. In function program verification, a correctness proof is done by comparing the function implemented by a program called a *code function* with its specification called an *intended function* [2] [3]. For this, each section of code is annotated with its intended function. If a section of code consists of only simple statements and control structures such as assignments, sequences and branches, its code function can be calculated directly and compared with its intended function. However, if it contains loops such as while statements, it is mostly impossible to calculate its code function directly, thus its proof is done by using the technique of proof-by-induction. Applying the inductive proof rule of while statements is in most case rather straightforward, but finding a correct induction hypothesis, the intended function of a while statement, is not and often is the most difficult part of the proof. And there is no systematic way of formulating a good intended function for a while statement, and programmers mostly relies on their intuitions, insights, or previous experiences to formulate one. Nevertheless, it is vital to formulate a correct intended function for a while statement, as an incorrect induction hypothesis will definitely fails an inductive proof.

One possible way to help the programmers find correct intended functions for while statements is to provide them with a catalog of sample while statements along with their intended functions. The samples in the catalog provide patterns of while statements and their intended functions that can be matched to one's own code and thus can be instantiated to derive one's own intended functions. If a while statement matches a code pattern in the catalog, its intended function will have a similar structure as that of the matched code in the catalog. That is, similarly-structured while statements have similarly-structured intended functions.

In this paper we describe our approach for identifying such patterns of while statements based on loop conditions and loop bodies, documenting them in a pattern catalog, and applying them to find intended functions of while statements. However, there are conflicting requirements for being a good pattern. A pattern should be as general as possible to be widely applicable and usable, but at the same time it should be as specific as possible to be meaningful in deriving an accurate intended function. We explain how we address these conflicting requirements. Like software design patterns that describe reusable design solutions to recurring problems in software design [4], our specification patterns also provide other benefits by allowing one (a) to capture and document program specification knowledge, (b) to support reuse in specification and boost one's confidence during program verification, and (c) to provide a vocabulary for communicating one's specifications and proofs. We explain our approach using functional program verification. However, we believe that our approach is equally applicable to other verification techniques such as Hoare-style axiomatic verification using pre- and post-conditions.

This paper is structured as follows. Section II provides a brief overview of functional program verification including the notation for writing intended functions. Section III describes the problem of finding and formulating intended functions of while statements. Section IV explains our approach for documenting and cataloging patterns of while statements along with their intended functions, and Section V illustrates some of our patterns by applying them to examples and provides a preliminary evaluation of our approach. Lastly Section VII concludes this paper with a concluding remark.

II. Functional program verification

Functional program verification is a program verification technique originated from the *Cleanroom Software Engineering* [1]. The main

idea behind functional program verification is to view and model a program as a mathematical function that maps one program state, an *initial state*, to another, a *final state*. The specification of a program called an *intended function* defines this mapping of states by describing the expected final state in terms of the initial state. Program verification is done by comparing the intended function of a program with its *code function*, the actual function implemented by the program. For this, each section of a program is documented with its intended function (see Figure 1).

```
1:  // f1: [r : = ∑i=o…a.length-1(a[i] > 0 ? 1 : 0)]
2:    // f2: [r, i := 0, 0]
3:      r = 0;
4:      int i = 0;
5:
6:    // f3:  [r, i := r + ∑j=i…a.length-1(a[j] > 0 ? 1 : 0), anything]
7:      while (i < a.length) {
8:        // f4: [r , i :=  a[i] > 0 ? r + 1 : r,  i + 1]
9:          // [r := a[i] > 0 ? r + 1 : r]
10:           if (a[i] > k)
11:             // [r := r + 1]
12:               r++;
13:       // [i := i + 1]
14:         i++;
15:     }
```

Figure 1: Code annotated with intended functions

An intended function is written using a *concurrent assignment* notation of the form $[x_1, x_2 \ldots, x_n := e_1, e_2, \ldots, e_n]$ stating that each x_i's new value in the final state is e_i evaluated concurrently in the initial state [2]. For example, the intended function f_1 in line 1 describes the behavior of whole code and asserts that the final value of *r* is the number of positive values contained in the array *a*. On the other hand, the intended functions f_2 and f_3 in lines 2 and 6 specify the sections of code in lines 3-4 and 7-15, respectively. In f_3, the keyword *anything* indicates that one doesn't care about the final value of the loop variable *i*. In this paper we write intended functions semi-formally by using the Java expression syntax and mathematical symbols such as $\sum$. There is also a formal notation for writing intended functions[4].

Once each section of code is annotated with its intended function, its correctness can be proved by comparing its code function with its intended function. This proof can be performed in a modular way by using the intended functions of lower level code in the proof of higher level code. For example, in order to prove the correctness of the code shown in Figure 1, we need to prove (a) the functional composition of f_2 and f_3 is correct with respect to f_1 and (b) both f_2 and f_3 are correctly implemented or refined by their code. If a section of code consists of only assignments, sequences, and branches, its correctness proof is often straightforward, as its code function can be directly calculated. For example, the code function for lines 3-4 is the same as its intended function, f_2. However, the proof of a loop such as a while statement is a bit involved, as there is no direct way of calculating its code function. It is done by using proof-by-induction [2]. For example, the correctness of code in lines 7-15 with respect to its intended function f_3 requires three sub-proofs: (a) termination of the loop, (b) a basis step of proving that when the loop condition doesn't hold an identity function (i.e., no state change) is correct with respect to f_3, and (c) an induction step of proving that when the loop condition holds the composition of f_4 and f_3 is correct with respect to f_3. The basis and induction steps are for when the loop makes no iteration and one or more iterations, respectively.

III. Intended Functions of While Statements

In order to apply functional programming verification effectively, it is important to formulate a correct intended function for the section of code to be verified. If the intended function is incorrectly formulated, the proof will fail even if the code is indeed correct. This is particularly true for the verification of loops such as while statements, as their proofs are done inductively and their intended functions become induction hypotheses (see Section II). With a wrong induction hypothesis, an inductive proof will fail.

However, formulating and defining a good intended function for a while statement is not easy. It is often the hardest part of formal program verification, and there is no systematic way of doing it. One difficulty is that a loop typically computes a more general function than the one needed. A loop is seldom used by itself in isolation but is preceded by an initialization, which together with the loop computes something useful. For example, the while statement in lines 7-15 of Figure 1 doesn't calculate the number of positive values contained in the array *a*, but when the loop variable *i* is set to 0 it does. A loop in isolation doesn't do a computation but completes it; an initialization (e.g., setting *i*) determines where the computation starts. An intended function of a while statement should be written in such a way that it captures the completion of a computation regardless of where the computation starts. It should be a correct generalization of the intended function for the code containing both the initialization and the loop, and at the same time it should be specific enough to capture the accurate result of the computation.

Formulating an intended function for a while statement requires a programmer's insight, practice, and experience [2]. The problem of finding an intended function for a while statement is similar to that of finding a loop invariant in Hoare logic. A loop invariant should be general enough to hold on each iteration of the loop and specific enough to lead to a post-condition when the loop terminates. Even if there is no known work done on systematically finding intended functions for loops, many researchers have studied this similar problem of finding loop invariants and proposed various static and dynamic techniques based on pre-conditions, post-conditions, loop executions, and theorem proving (cf., [7] [8] [9]).

IV. Patterns of While Statements

One way to figure out a correct intended function of a while statement is to look at other while-loops that have similar code structures. If two while loops have similar code structures, their intended functions are likely to have similar structures too [2]. Therefore, if we know the intended function of one, we may be able to derive that of the other from the known one. For this, we can develop patterns of while loops along with their intended functions based on the code structures of while loops including loop conditions and loop bodies, and these patterns can be used as a reference for formulating an intended function for a while loop (see Figure 2). For this pattern-based approach to work effectively, we need to identify and accumulate a large number of patterns to cover a wide range of while loops appearing in application code. And each pattern should be as general as possible to be widely applicable to loops written in many different ways. At the same time it should be as specific as possible to derive an accurate intended function when applied to a particular loop. In any pattern-based approach, properly documenting patterns is crucial. Each pattern should be documented in such a way that it is easy to determine its applicability, to instantiate it for a

particular application, and to derive an actual intended function from it. Patterns need to be classified and organized to be presented in a *pattern catalog* that can be easily looked up and matched for by programmers. Below we explain how we address these requirements for our patterns.

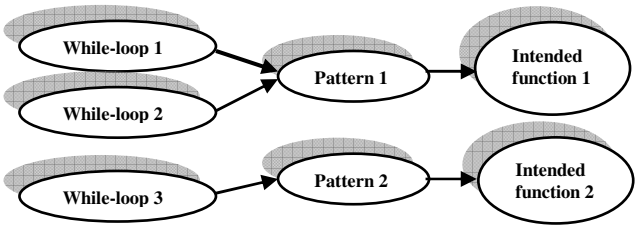

Figure 2: Pattern-based identification of intended functions

A. Pattern Documentation

We document our patterns using a format similar to that of software design patterns [4]. Each pattern has a name, purpose, description, structure, applicability, variations, related patterns, and examples. Figure 3 shows one of the simplest patterns that we identified and documented in this format. A pattern has a *name* to uniquely identify it. Then, its *purpose* is stated briefly. The *description* section explains the pattern and is followed by the *structure* of the code along with its intended function. The structure is given as a skeletal annotated code; as shown, the body of a while loop can be abstracted to an intended function to be applicable to a wide range of implementation variations. The *applicability* section lists different contexts in which the pattern can be applied. A pattern can have *variations* and *related patterns*. Lastly the *examples* section shows sample loops matching the specified pattern.

The pattern depicted in Figure 3 is named *Indexed Accumulating*. It describes a while loop that iterates over the elements of a sequence using an index and accumulates them by using a binary accumulation operator such as +, *, and string concatenation. In the loop body abstracted to an intended function, the element of the sequence *s* at the current index *i* (i.e., *s@i*) is accumulated to the result variable *r* using an accumulator operator ⊕ (i.e., $r := r \oplus s@i$), and the index variable *i* is set to a new value *E*(*i*), an expression written in terms of *i*. The loop iterates as long as the loop condition *B*(*i*), a Boolean expression written in term of *i*, holds. The intended function of this loop states that the final of the result variable *r* is its initial value accumulated or combined with all the elements of *s* starting at index *i* to index *N*, where *N* is the value of *i* just before the loop condition becomes false. This pattern is applicable when the sequence is an array, a string, and an index-based collection like a Java List class. It has several variations including Conditional Accumulating in which an element is accumulated only if it satisfies a certain condition, e.g., being a positive value. If a sequence or collection provides an iterator (cf. Iterator pattern in [4]), its values can be merged or accumulated by using its iterator operations rather than indexing, and the Iterated Accumulating pattern is for such loops.

B. Sample Patterns

To identify patterns, we studied a wide range of while loops from several different sources including computer programming textbooks, class programming assignments and projects, and well-known open source software. Through this study we were able to identify patterns of recurring while loops and documented them as specification patterns by generalizing their source code structures and formulating their intended functions. Below we describe a few representative patterns that we identified and documented.

Name: Indexed Accumulating
Purpose: Accumulate elements of a sequence
Description: A loop combines the values of a sequence to a single value by using various accumulation operations such as addition, multiplication, and concatenation. An index is used to iterate over the elements of a sequence. The result is of the same type as that of the elements of the sequence.
Structure:

```
[r, i := r ⊕ ∑⊕ j=i..N s@j, anything]
while (B(i)) {
    [r, i := r ⊕ s@i, E(i)]
}
where
 s: sequence whose elements are accumulated
 r: result variable accumulating elements of s
 i: index and loop variable
 s@i: i-th element of s
 ⊕: accumulation operator such as +, -, and *
 B(i): Boolean expression with a variable i
 E(i): expression with a variable i
 N: last i prior to loop termination such that B(i)
```

Applicability: arrays, strings, indexable collections, etc.
Variations: Conditional Accumulating, ...
Related patterns: Iterated Accumulating, …
Examples:

```
[r, i := r + ∑ j=i..a.length-1 a[i], anything]
 while (i < a.length) {
    [r, i := r + a[i], i + 1]
     r = r + a[i];
     i++;
 }
```

Figure 3: Indexed Accumulating

One related pattern of the Indexed Accumulating pattern described previously is a pattern named Indexed Conditional Counting (see Figure 4) As hinted by its name, it represents a loop that counts the number of elements contained in a sequence that meets a certain condition. Its structure is very similar to that of the Indexed Accumulating except that instead of accumulating the elements of a sequence it accumulates 1's for elements that satisfies a certain condition, thus counting the occurrences of elements in a sequence that satisfies the condition. In the pattern, the notation *C*(*s@i*) denotes a Boolean expression that checks if the *i*-th element of the sequence *s* satisfies a certain condition; if there is no such a condition imposed, the loop calculates the cardinality of the sequence.

```
[r, i := r + ∑ j=i..N C(s@j) ? 1: 0, anything]
while (B(i)) {
    [r, i := r + (C(s@i) ? 1: 0), E(i)]
}
```

Figure 4: Indexed Conditional Counting

Another recurring pattern of while loops is searching for an element in a collection. For example, a while loop may look for any negative value contained in a list. We named this pattern Indexed

Searching (see Figure 5). The intended function states that the final value of the result variable *r* is the element of the sequence *s* at index *j*, denoted by *s@j*, if the element at index *j* satisfies the searching condition *C,* i.e., *C*(*s@j*); if there is no such a *j*, it is the initial value of *r*. Note that the loop condition *B*(*i*, *r*) may refer to the result variable *r* to allow an early termination of the loop, e.g., as soon as an element is found.

```
[r, i := (∃j=i..N C(s@j)) ? (s@k s.t. k∈i..N ∧C(s@k)) : r, anything]
while (B(i, r)) {
   [r, i := C(s@i) ? s@i : r, E(i)]
}
```

Figure 5: Indexed Searching

A loop is also frequently used to select or collect elements from a collection. We documented this use of while loop as a family of patterns, and one particular pattern named Iterated Collecting is shown in Figure 6. This particular pattern is for selecting elements of a collection by accessing them using an iterator and transforming them to construct a new collection. Since there are many different implementations of iterators, we abstract away from implementation details of iterators in our pattern documentation by introducing abstract iterator operations such as *hasNext*, *current*, and *advance*. There are several variations of this pattern, e.g., collecting only those elements that meets a particular selection criterion and selecting elements without transforming them.

```
[r, i := r ∪ {e ∈ Ci • E(e)}, anything]
while (B(i.hasNext())) {
   [r, i := r ∪ {E(i.current())}, i.advance()]
}
where
 ∪: collection merge operation such as union, concatenation, etc.
 i: iterator of the collection whose elements are to be collected.
 i.hasNext(): true only if the iterator i has more elements
 i.current(): current element of the iterator i
 i.advance(): move to the next element of the iterator i
 Ci: all elements available from the iterator i
```

Figure 6: Iterated Collecting

```
[r, i := r ⊕ ∑⊕ j=i..N E1(j), anything]
while (B(i)) {
   [r, i := r ⊕ E1(i), E2(i)]
}

// Instantiated with bindings
//  × for ⊕, ∏ for ∑⊕, 1 for N, i > 0 for B(i), i for E1(i),
//  i – 1 for E2(i)
// [r, i := r × i × (i-1) × (i-2) × ... × 1, anything]
while (i > 0) {
   // [ r, i = r × i, i – 1]
     r = r * i;
     i--;
```

Figure 7: Variation of the Accumulating pattern

All the patterns introduced and described so far are for accessing and manipulating elements of collections such as arrays, strings, sequences, streams, and files. However, a loop doesn't need to access or manipulate a collection, and another typical use of it is to iterate an indefinite number of times. A while statement, for example, can be used to calculate the factorial of a positive number. Figure 7 shows a variation of the Accumulating pattern described earlier along with its instantiation for factorial code. In fact, it is a generalization of the Indexed Accumulating pattern in that each occurrence of reference to the *i*-th element of the sequence, *s@i*, is abstracted and generalized to an expression *E*(*i*). We will discuss more on pattern generalization and specialization in the following subsection.

C. Pattern Classification and Hierarchy

While analyzing many different while loops, we soon learned that the structure of a pattern is determined by three factors: (a) how the value to be manipulated is obtained, (b) how the value is manipulated, and (c) how the termination of the loop is determined. These three factors are mostly orthogonal, and thus most combinations of them produce new patterns (see Figure 8). For example values can be retrieved from collections like arrays, strings, streams, and files using indices, iterators, or in ad-hoc fashions, or they can be created on the fly without retrieving stored ones. There are many different manipulations of values possible, e.g., accumulating (conditionally or unconditionally and with or without transformation), searching, counting, selecting, and collecting. The loop conditions may be written in terms of indices, iterators, values being manipulated, and others. Therefore, we can define our patterns compositionally by picking up one particular possibility for each of these three factors. For example, the Indexed Searching pattern is a composition of an index-based acquisition, a search manipulation, and an index-based termination.

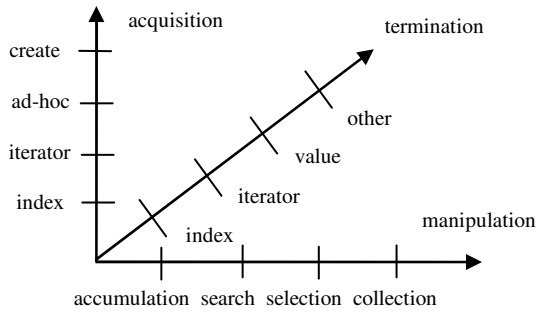

Figure 8: Orthogonal factors of patterns

As mentioned previously, one key requirement of patterns is to make them as general as possible and at the same time as specific as possible. This is to make patterns as widely applicable as possible and at the same time to derive accurate and detailed intended functions upon their applications. To address this requirement we generalized and specialized patterns to produce a pattern hierarchy. The idea is to have abstract or general patterns to cover a wide range of while loops but with coarse-grained intended functions. Concrete or specialized patterns will cover a limited range of while loops but will provide more accurate and detailed intended functions. In the previous subsection, for example, an index-based access of elements was denoted by an expression s@*i*, where *s* is a sequence and *i* is an index, and an iterator-based access was denoted by an expression *i.current*(). We can unify these two expressions to come up with a more abstract expression *E*(*i*) and use this abstract expression to define a pattern, thus resulting in a more abstract, general, and widely applicable pattern. Applying such a pattern, however, requires more

work, as a correct instantiation of an abstract expression like $E(i)$ may not be straightforward for both matching a pattern and deriving an intended function from the matched pattern.

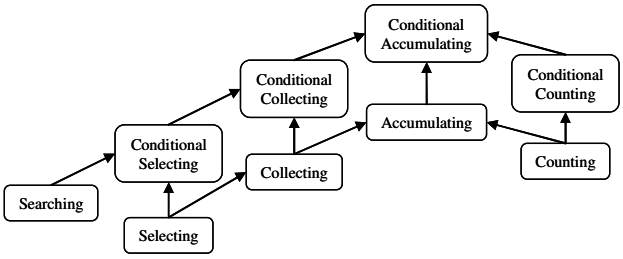

Figure 9: Pattern hierarchy

Figure 9 shows a simplified version of a pattern hierarchy focusing on the manipulation of values, not their acquisitions or loop termination. The *Selecting* pattern, for example, is a special *Conditional Selecting* without any imposed condition as well as a special *Collecting* without any transformation of values. As mentioned earlier and shown in the figure, a higher level pattern such as *Conditional Accumulating* is more general and thus has an intended function applicable to a wide range of while loops. A lower level pattern such as *Selecting* is more specific and thus has a more detailed intended function with a narrow scope of applications. A general guideline is to match patterns starting from the root of the tree and move downward to find as specific pattern as possible.

V. Pattern Applications

We conducted a preliminary experiment to evaluate the effectiveness of our patterns by applying them to industrial strength open-source code. We chose Apache HTTP Server 2.0.65 that has about 486 C files with over 1500 while loops [10]. We picked up a dozen different while loops from the Apache source code and applied our patterns. For pattern matching we used a decision tree similar to the one shown in Figure 10. The decision tree allows us not only to perform pattern matching systematically and semi-automatically but also to identify general patterns first and then move toward more specific ones. Below we describe a few simple but interesting while loops from the Apache source code to illustrate applications of our patterns.

The most common use of while statements in the Apache source code is to manipulate pointer-based data structures such as linked lists. Shown below is one such a while loop that is simple but shows an interesting aspect of the application of our patterns. It traverses a linked list pointed to by *f* and changes the *r* field of each node if its current value is equal to *from*.

```
while (f) {
   if (f->r == from)
      f->r = to;
   f = f->next;
}
```

Even if the loop itself is very simple, specifying its behavior is a bit involved because it may mutate not only a single node pointed to by *f* but also potentially all the nodes reachable from *f*. Thus, its intended function needs to capture the side effect caused to the whole list, not just to a single node denoted by the pointer *f*. For this, we introduce a notation to denote the whole list and manipulate a pointer-based list data structure abstractly.

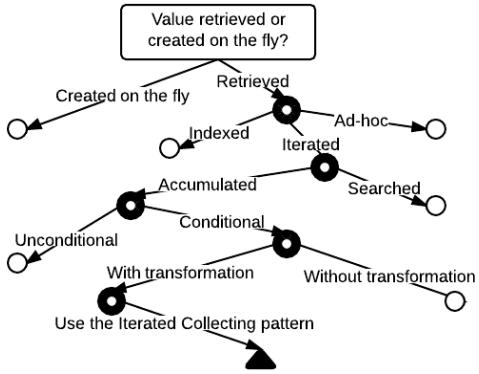

Figure 10: Using patterns

$L_{<f}$: list consisting of all the nodes preceding f; whole list if f is null
$L_{f>}$: list consisting of f and all the trailing nodes; empty if f is null
$\langle\rangle$: list comprehension, e.g., $\langle f\rangle$ for a singleton list consisting of f
+: list concatenation, e.g., $L_{<f} + \langle f\rangle$
$f\{r := e \text{ if } b\}$: node f with its *r* field set to *e* if *b* is true

Using this notation we first calculate and document the intended function of the loop body as follows.

```
while (f) {
   // [L<f, f := L<f + ⟨f{r := to if r == from}⟩, f->next]
      if (f->r == from)
         f->r = to;
      f = f->next;
}
```

Note that the intended function of the loop body can also be written as [f->r, f := (f->r == from) ? to : f->r, f->next]. However, this formulation doesn't capture the side effect of the statements in terms of the whole list and thus will make it difficult to prove the correctness of the whole loop. Once the intended function of the loop body is formulated and documented, we can match the annotated code to one of the patterns. Using the decision tree mentioned earlier we can match it to the Iterated Collecting pattern (see Figure 10). Although this matching doesn't seem possible at first, it should be apparent with the following unification of iterator operations.

Pattern	Code
i.current()	f
i.advance()	f = f->next
i.hasNext()	!f

Once a matching pattern is found, we can instantiate its intended function to derive the intended function for the code as follows, where the notation $I(x/y)$ denotes replacing every free occurrence of y in the intended function I with an x.

$[r, i := r \cup \{e \in C_i \bullet E(e)\}, \text{anything}]\ (r/L_{<f}, i/f, L_{f>}/C_i, +/\cup, \langle\rangle/\{\},$
$E(e)/e\{r := \text{to if } r == \text{from}\}) \equiv$
$[L_{<f}, f := L_{<f} + \langle n \in L_{f>} \bullet n\{r := \text{to if } r == \text{from}\}\rangle, \text{anything}]$

The derived intended function states that every node reachable from *f* now has a new final value (*to*) for its *r* field if its initial value is *from,* and it matches our informal understanding of the loop.

Another while loop from the Apache source code is shown below. It iterates over the nodes of a list to check if there is a node with a particular name. The code matches our Iterated Searching pattern,

and the matching produces the intended function annotated in the code. The Iterated Searching pattern is similar to the Indexed Searching pattern described in the previous section except that elements are accessed using an iterator. As before we need to refer to the whole list, and for this we introduce a special notation L_{filter} to denote the list consisting of the node pointed to by *filter* and all the nodes reachable from it.

```
/* [found, filter := (∃n∈ L_filter• f(n)) ? true: found, anything] where
 *   L_filter = list consisting of filter and all nodes reachable from it
 *   f(n) = !strcasecmp(name, n->frec->name) */
while (!found && filter) {
   // [found, filter  := f(filter) ? true : found, filter->next]
     if (!strcasecmp(name, filter->frec->name))
        found = true;
     filter = filter->next;
}
```

In the Apache source code, we also found while loops that refer to arrays. The following is one such a loop, and its loop body is annotated with its intended function written using a conditional concurrent assignment of the form $[B_1 \rightarrow A_1 \mid \ldots B_n \rightarrow A_n]$ that specifies different functions (A_i's) based on conditions (B_i's)[2].

```
while (name[i] != '\0') {
   /* [C(i) → i := i + 2
    *  | ¬C(i) → w, i, name[w-1] := w + 1, i + 1, name[i]]
    *  where C(i) is the if condition below. */
     if (name[i] == '.'  && IS_SLASH(name[i+1])
        && (i == 0 || IS_SLASH(name[i – 1])))
        i += 2;
     else
       name[w++] = name[i++];
}
```

Its intended function is not obvious, but our decision tree and patterns can guide us to it. For example, the values to be manipulated in the loop are *retrieved* from an array using an *index*, selected on a *condition*, and stored *without being transformed*. This leads us to the Indexed Conditional Selecting (or its generalizations) as a possible pattern. The pattern's intended function and our insight on the code lead us to the following intended function, where $s[i..j]$ denotes a substring of s from index i to j, inclusive.

```
[name, i, w := name[0..w-1] + shifted name[w..], anything,
  w + num of shifted chars]
```

For more rigorous formulation, one need to state precisely what one means by a *shifted name*[*w*..]; informally, it's the suffix of *name* starting from index *w* in which the characters of *name* starting from index *i* that doesn't match the specified patterns (e.g., "./") were shifted over those characters matching the patterns.

Even if our experiment is limited in the number of while loops, the applications and the implementation languages that we considered, it showed a promising result. We often were able to systematically derive very detailed and precise intended functions. Most non-trivial loops, however, required a varying degree of insight and work to come up with accurate intended functions for them. Nevertheless, the pattern decision tree helped us to analyze the code systematically and the matched patterns helped us find correct intended functions for these loops by providing skeletal intended functions. Our patterns also detected certain mistakes that we made when formulating intended functions for both the loop body and the whole loop—e.g., documenting side-effect caused only to a single node, not to the whole data structure. In fact we made such mistakes several times during this experiment, and our patterns exposed and revealed them. We also learned a few shortcomings of our patterns through this experiment. Our patterns, for example, don't capture a certain common use of while loops very well, e.g., mutating the collection data structure that is being iterated over. These loops can be certainly abstracted to selecting or collecting patterns as done in this paper, but more direct and concrete patterns would be preferred and appreciated by the users of our patterns. Some of our patterns may be further specialized to address language-specific features and constructs such as C/C++ pointers and pointer-based data structures.

VI. CONCLUSION

In this paper we proposed specification patterns to address the problem of formulating specifications of loop statements such as while statements, which is recognized as one of the most difficult part of formal program verification. Our approach was initially inspired by software design patterns [4], however the key difference and thus contribution of our patterns compared with design patterns and other specification patterns (e.g., [6]) is that our specification patterns are *compositional* and *hierarchical*. Each pattern consists of three orthogonal components—value acquisition, value manipulation, and loop condition—and thus is assembled by selecting an appropriate combination of these building blocks. Our patterns are classified into a pattern hierarchy. A generalized pattern is applicable to a wide range of loops, but its specification is more abstract. A specialized pattern, on the other hand, is more specific with limited applicability but provides a more accurate specification. The pattern hierarchy allows one to match patterns starting from more general ones to more detailed. The work reported in this paper is an on-going research, and thus the number of patterns identified and documented is limited. Nevertheless, a preliminary experiment showed a promising result in that the patterns were able to derive specifications for a representative set of while loops found in well-known open source code.

REFERENCES

[1] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering*. Addison Wesley, Feb. 1999.

[2] A. Stavely, *Toward Zero Defect Programming*. Addison-Wesley, 1999.

[3] Y. Cheon, *Functional Specification and Verification of Object-oriented Programs*, Department of Computer Science, The University of Texas at El Paso, El Paso, TX, Technical Report 10-23, Aug. 2010.

[4] Y. Cheon, C. Yeep, and M. Vela, The CleanJava Language For Functional Program Verification, *International Journal of Software Engineering*, 5(1):47-68, Jan. 2012.

[5] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, Patterns in Property Specifications for Finite-state Verification, *21st International Conference on Software Engineering*, pages 411-420, May, 1999.

[7] C. A. Furia and B. Meyer, Inferring Loop Invariant Using Postconditions, *Fields of Logic and Computation, volume 6300 of Lecture Notes in Computer Science*, pages 277-300, Springer, 2010.

[8] R. Sharma, et al., A Data Driven Approach for Algebraic Loop Invariants, *ESOP, volume 7792 of Lecture Notes in Computer Science,* pages 574-592. Springer, 2013.

[9] K. R. M. Leino and F. Logozzo, Loop Invariants on Demand, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05),* pages 119-134, Nov. 2005.

[10] The Apache Software Foundation, *Apache HTTP Server Project*, available from http://httpd.apache.org.