12-2015

A Systematic Derivation of Loop Specifications 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-15-90

A Systematic Derivation of Loop Specifications Using Patterns

Aditi Barua and Yoonsik Cheon

TR #15-90
December 2015

Keywords: formal proof, functional program verification, intended function, program specification, specification pattern, while statement

1998 CR Categories: D.2.4 [*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.

A Systematic Derivation of Loop Specifications Using Patterns

Aditi Barua[1] and Yoonsik Cheon[2*†]

[1]*Center for Institutional Evaluation, Research and Planning, The University of Texas at El Paso, El Paso, Texas 79968-0518, U.S.A.*
[2]*Department of Computer Science, The University of Texas at El Paso, El Paso, Texas 79968-0518, U.S.A.*

SUMMARY

Any non-trivial program contains loop control structures such as while, for and do statements. A formal correctness proof of code containing loop control structures is typically performed using an induction-based technique, and oftentimes the most challenging step of an inductive proof is formulating a correct induction hypothesis. An incorrectly-formulated induction hypothesis will surely lead to a failure of the proof. In this paper we propose a systematic approach for formulating and driving specifications of loop control structures for formal analysis and verification of programs. We explain our approach using while loops and a functional program verification technique in which a program is viewed as a mathematical function from one program state to another. The most common use of loop control structures is to iterate over a certain sequence of values and manipulate it, one value at a time. Many loops exhibit certain common flavors or patterns, and similarly-structured loops have similarly-structured specifications. Our approach is to categorize and document the common flavors or usage patterns of loop control structures as reusable specification patterns. One key idea of our pattern specification is to promote manipulation of individual values to the whole sequence iterated over by a loop. Our patterns are compositional and can be organized into a pattern hierarchy. A catalog of loop specification patterns can be a good resource for systematically formulating and deriving specifications of loops. Indeed, our case study indicates that our patterns are applicable to a wide range of programs from systems programming to scientific and business applications.

KEY WORDS: formal proof, functional program verification, intended function, program specification, specification pattern, while statement.

1. INTRODUCTION

In the functional program verification method, a program is viewed as a mathematical function from one program state to another, and a correctness proof of a program is done by comparing the function implemented by the program, called a *code function*, with its specification called an *intended function* [7, 35, 36]. For the verification, each section of code is annotated with its intended function. If a section of code consists of only simple statements or control structures such as assignment statements, conditional statements, and sequences of simple statements, its code function can be calculated directly from the code and then compared with the intended function. However, if the code contains loop statements like while statements, it may be impossible to calculate its code function directly from the code, thus its proof should be done by using a technique based on induction. Dealing with loops is the most difficult part of program analysis as well as formal verification [25]. Applying a proof-by-induction technique involves formulating an induction hypothesis and proving its truth both for basis cases and inductive steps. In general, proving the induction hypothesis can be done systematically or even semi-automatically by symbolically

*Correspondence to: Yoonsik Cheon, Department of Computer Science, The University of Texas at El Paso, El Paso, Texas 79968-0518, U.S.A.
†Email: ycheon@utep.edu

executing statements and recording their side-effects in a table called a *trace table* to calculate intended functions [35, 36]. However, finding a correct induction hypothesis of a loop—e.g., a candidate or likely intended function of a while loop—is not, and it is the most difficult step of an inductive proof [35]. This is because there is no simple rule or systematic way of formulating a good intended function for a loop statement, and thus programmers rely on their intuitions, insights, skills, and experiences. Nevertheless, it is crucial to come up with a good intended function for a loop statement, for an incorrect induction hypothesis will fails an inductive proof.

One possible way to help programmers find correct or likely intended functions for loops is to provide them with a catalog of sample, representative loops along with their intended functions [4]. The samples in the catalog provide patterns of loops along with their intended functions that can be matched to and instantiated for particular occurrences of loops in one's code. If a particular loop matches a pattern in the catalog, its intended function is likely to have a similar structure as that of the matching pattern. Our approach is based on the observation that many loops exhibit certain common flavors or patterns, and similarly-structured loops have similarly-structured intended functions. For any pattern-based approach to be useful in practice, however, the choice and the variety of patterns are crucial. There is also a conflicting requirement for specification patterns. A good specification 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, detailed intended function when applied and instantiated. Like software design patterns that describe reusable design solutions to recurring problems in software design [20], our loop specification patterns also provide other benefits by allowing one (a) to capture and document program specification knowledge, (b) to support reuse in program specification and boost one's confidence in the analysis and verification of programs, and (c) to provide a vocabulary for communicating formal program specifications and proofs.

We explain our pattern-based approach for systematically deriving likely loop specifications for functional verification of programs [5]. A recent study shows that, among the three main loop control structures (for, while, and do statements) in C, C++, and Java, the most frequently used is the for statement [37]. However, since the for statement can be viewed as a syntactic sugar of the while statement, we use the while statement as a representative loop control structure to explain our approach. In fact, the proof rule of the for statement is a specialization of that of the while statement [35]. We identified and documented a number of specification patterns to capture the common use of while loops, and some of the patterns are specializations or sub-patterns of other more general ones [4]. The most common use of while loops is to iterate over a certain sequence of values and manipulate it, one value at a time. One of the key ideas of our loop pattern documentation is to promote the manipulation of individual values to the whole sequence iterated over by a loop. For this, we also invented a conceptual framework for analyzing while loops systematically. The framework consists of four different, orthogonal analysis dimensions, including one for analyzing the manipulation of individual values iterated over by a loop, making our patterns compositional. We used the framework to identify and classify different while loops along with their intended functions. The documented patterns are language-neutral in that they can be applied to a wide range of programming languages, from imperative, procedural languages to object-oriented languages. For example, the patterns can be matched to while loops that iterate over different implementations of index-based collections like arrays, strings, and sequences, as well as iterator-based collections like linked lists and pointer or reference-based collection data structures commonly found in popular programming languages such as C, C++, and Java. The documented patterns have skeletal loop code, consisting of loop conditions and bodies, as well as skeletal intended functions. The cataloged pattern can be used to derive intended functions of while loops by first matching the loops to loop patterns and then instantiating the corresponding skeletal intended functions. Once candidate or likely intended functions are formulated and written, the correctness of the loops can be proved rigorously or formally using the functional program verification technique in which a program is viewed as a mathematical function from one program state to another [7, 35, 36]. We also suggest a step-by-step process for applying the cataloged patterns to derive intended functions systematically

and semi-automatically. In a case study, we applied our patterns to source code of several open-source projects by examining and analyzing more than 100 while loops. Our findings are very promising in that our patterns are applicable to a wide range of programs from systems programming to scientific and business applications, covering 96% of loops examined. Even though we explain our approach using functional program verification, we believe its key ideas be equally applicable to other program specification and verification techniques such as Hoare-style axiomatic approaches.

The rest of this paper is organized as follows. In Section 2 we provide a brief overview of functional program verification, including the notation for writing intended functions, formal correctness proof of while loops, and the challenge of finding likely intended functions for while loops. In Section 3 we explain our approach for documenting and cataloging patterns of while loops and their intended functions. We first describe a new conceptual framework for analyzing while loops systematically. The framework is used to identify and classify different loop patterns. We then describe in detail several representative loop patterns documented using a format similar to that of software design patterns. In Section 4 we suggest a step-by-step process for applying our documented patterns. We show sample applications of two of our documented patterns by following the suggested process. In Section 5 we evaluate our approach and patterns and summarize our findings along with lessons learned. In Section 6 we mention few broadly related work, including loop invariants, and we conclude this paper with a concluding remark in Section 7.

2. FUNCTIONAL PROGRAM VERIFICATION

In the late 70s, Harlan Mills and his colleagues at IBM developed an approach to software development named *Cleanroom Software Engineering* [28, 30, 33], Its name was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication, and the method reflects the same emphasis on defect prevention rather than defect removal. Special methods are used at each stage of the software development—from requirement specification and design to implementation—to avoid errors. In particular, it uses specification and verification, where verification means proving mathematically that a program agrees with its specification. Cleanroom is a lightweight, or semi-formal, method and tries to verify the correctness of a program using a technique called *functional program verification* [7, 35, 36]. The technique requires a minimal mathematical background by viewing a program as a mathematical function from one program state to another and by using equational reasoning based on sets and functions. The specification of a program called an *intended function* defines this mapping of states by describing the expected final state in terms of an initial state [35]. In essence, the functional verification involves (a) calculating the function computed by code called a *code function* and (b) comparing it with the intention of the code also written as a function, an intended function. For this, the behavior of each section of code is documented, as well as the behavior of the whole program. The documented behavior is the specification to which the correctness of a program is verified.

2.1. Programs As Functions

An execution of a program produces a side-effect on a program state by changing the values of some state variables such as program variables. In functional program verification, a program execution is modeled as a mathematical function from one program state to another, where a program state is a mapping from state variables to their values. For example, consider the following code snippet that swaps the values of two variables x and y.

```
x = x + y;
y = x − y;
x = x − y;
```

Its execution can be modeled as a mathematical function that, given a program state, produces a new state in which x and y are mapped to the initial values of y and x, respectively. The rest

of the state variables, if any, are mapped to their initial values; their values remain the same. This is a more direct way of describing computations than the assertions used in Hoare-style axiomatic verification, which state facts about values of variables.

A succinct notation, called a *concurrent assignment*, is used to express these functions by only stating changes in an input state [3, 35, 36]. A concurrent assignment is written as $[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$ and states that each x_i's new value is e_i, evaluated concurrently in the initial state, i.e., the input state or the state just before executing the code. The value of a state variable that doesn't appear in the left-hand side of a concurrent assignment remains the same. For example, the function that swaps two variables, x and y, is written as $[x, y := y, x]$. The concurrent assignment notation can be used to express both the actual function computed by a section of code, a code function, and one's intention for the code, an intended function.

2.2. Correctness Verification

The verification method is quite different than the method of axiomatic verification. It is based on functional semantics and on the reduction of software verification to ordinary mathematical reasoning about sets and functions as directly as possible. The correctness of code is verified by comparing its code function to its intended function; verification means showing that the code function computes the result predicted by the intended function. A program, or a section of code, with an intended function f is correct if it has a code function p such that:

- The domain of p is a superset of the domain of f, i.e., $\text{dom}(p) \supseteq \text{dom}(f)$. The program may accept more values than what its specification says.
- For every x in the domain of f, p maps x to the same value that f maps to, i.e., $p(x) \equiv f(x)$ for $x \in \text{dom}(f)$. For each value allowed by its specification, the program should produce the same value as stated in the specification

It is also said that p is a *refinement* of f, denoted by $p \sqsubseteq f$. For correctness verification, an intended function is written for each section of the code to be verified. For example, Listing 1 show an annotated code snippet that counts the number of positive values contained in an array. An indentation is used to indicate the region of code that an intended function annotates. For example, the intended function f_0 in line 1 describes the behavior of the whole code and states that the final value of r is the number of positive values contained in the array a. The intended functions f_1 and f_2 in lines 2 and 6 specify the sections of code in lines 3–4 and 7–15, respectively. In f_3, the word *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 using Java expressions and well-known mathematical notations like Σ. There is also a formal specification language for writing intended functions [9].

Once each section of code is annotated with its intended function, its correctness can be proved by comparing its code function and intended function. A proof can be done 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 above, one needs to prove (a) the function composition of f_1 and f_2 is correct with respect to f_0 and (b) both f_1 and f_2 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, for its code function can be calculated directly from the code using tools such as *trace tables* facilitating symbolic execution of statements and functions [35, 36]. For example, the code function for lines 3–4 is exactly the same as its intended function, f_1. However, a correctness proof of a loop such as a while loop is generally more involved because there is no direct way of calculate its code function. It is done by using a proof-by-induction technique [35, 36]. For example, the correctness of code in lines 7–15 with respect to its intended function f_2 requires three sub-proofs: (a) termination of the loop, (b) a basis step proving that when the loop condition doesn't hold, an identity function (i.e., no state change) is correct with respect to f_2, and (c) an induction step proving that when the loop condition holds, function composition of f_3 (intended function of the loop body) and f_2 is correct with respect to f_2. The basis and induction steps are for when the loop makes no iteration and one or more iterations, respectively. Therefore, verification of the above code requires discharging the following four proof obligations.

Listing 1. Code annotated with intended functions

```
1  // f0: [r := Σ_{j=0..a.length−1}(a[i] > 0 ? 1 : 0)]
2   // f1: [r, i := 0, 0]
3   r = 0;
4   int i = 0;
5
6   // f2: [r, i := r + Σ_{j=i..a.length−1}(a[j] > 0 ? 1 : 0), anything]
7   while (i < a.length) {
8    // f3:  [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  }
```

1. $f_1; f_2 \sqsubseteq f_0$, i.e., a proof that f_1 followed by f_2 is a refinement of f_0, where the symbol ";" denotes the forward function composition.
2. Refinement of f_1, i.e., correctness of f_1's code.
3. Refinement of f_2, which requires the following three sub-proofs.
   (a) Termination of the loop
   (b) Basis step: $\neg(i < a.length) \Rightarrow I \sqsubseteq f_2$, where I denotes an identity function.
   (c) Induction step: $i < a.length \Rightarrow f_3; f_2 \sqsubseteq f_2$
4. Refinement of f_3, i.e., correctness of the loop body

Below we prove the correctness of the while loop by discharging its three proof obligations listed above.

1. *Termination of the loop*. The intended function of loop body (f_3) state that i is incremented by 1 on each iteration of the loop, and thus i will eventually become equal to ***a.length***, at which time the loop terminates. That is, ***a.length - i*** is a *loop variant* whose value is decreased on each iteration of the loop, thereby ensuring its termination.
2. *Basis step*: $\neg(i < a.length) \Rightarrow I \sqsubseteq f_2$, where I is an identity function. If we assume $\neg(i < a.length)$, we have the following.

$$\begin{aligned} f_2 &= [r, i := r + \Sigma_{j=i..a.length-1}(a[j] > 0\ ?\ 1 : 0), anything] \\ &\equiv [r, i := r, anything] \ \ (\because\ i \geq a.length) \\ &\sqsupseteq [r, i := r, i] \\ &\equiv I \end{aligned}$$

   Therefore, $\neg(i < a.length) \Rightarrow I \sqsubseteq f_2$.
3. *Induction step*: $i < a.length \Rightarrow f_3; f_2 \sqsubseteq f_2$.

$$\begin{aligned} f_3; f_2 &= [r, i := a[i] > 0\ ?\ r + 1 : r, i + 1]\ ; \\ &\quad\ [r, i := r + \Sigma_{j=i..a.length-1}(a[j] > 0\ ?\ 1 : 0), anything] \\ &\equiv [r, i := (a[i] > 0\ ?\ r + 1 : r) + \Sigma_{j=i+1..a.length-1}(a[j] > 0\ ?\ 1 : 0), anything] \\ &\equiv [r, i := r + (a[i] > 0\ ?\ 1 : 0) + \Sigma_{j=i+1..a.length-1}(a[j] > 0\ ?\ 1 : 0), anything] \\ &\equiv [r, i := r + \Sigma_{j=i..a.length-1}(a[j] > 0\ ?\ 1 : 0), anything] \\ &\equiv f_2 \end{aligned}$$

Therefore, $i < a.length \Rightarrow f_3; f_2 \sqsubseteq f_2$.

In functional program verification, a proof is often straightforward because one can calculate code functions and compare them against intended functions. Although one needs to use such techniques as case analysis and induction depending on the control structures used as shown above in the proof of a while loop, carrying out a proof itself is essentially the same as that of a block of sequential statements. As shown above, unlike an axiomatic approach, functional verification supports forward reasoning, which is intuitive and natural in that it matches the way programmers reason about the correctness of programs informally.

2.3. Intended Functions of While Loops

In order to apply functional programming verification effectively, it is crucial to formulate a correct intended function for the section of code to be verified. If the intended function itself is incorrect, the proof will fail even if the code is indeed correct. This is particularly true for proofs of loop control structures such as while statements, as their proofs are done inductively and their intended functions become induction hypotheses (see Section 2.2). An inductive proof will fail with an incorrect induction hypothesis.

However, formulating and writing a candidate or likely intended function for a while loop is challenging. It is often the hardest part of formal program verification. There is no simple rule to calculate it nor a systematic way of doing it. One difficulty is that a loop typically computes a more general function than needed for a given task [35, Section 4]. A while 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 loop in lines 7–15 of Listing 1 doesn't count the number of positive values contained in the whole array a. It performs a more general function, counting the number of positive values in a starting from the index i. When the loop variable i is set to 0, however, it does count the whole array. In a sense, a loop in isolation doesn't do a computation but completes it. An initialization, e.g., setting i to 0, determines where the computation starts. An intended function of a while loop 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 of a while loop requires a programmer's insight, practice, skill, and experience. The challenge of finding a likely intended function of a while loop is similar to that of finding a likely loop invariant in an axiomatic approach. A *loop invariant* is a property that holds before and after each repetition of a loop and is essential for understanding the effect of a loop and proving its properties [22]. A loop invariant should be general enough to hold during each iteration of the loop and specific enough to lead to a postcondition when the loop terminates. Many researchers have studied the problem of finding loop invariants and proposed various static and dynamic techniques (see Section 6).

3. WHILE LOOP PATTERNS

One way to figure out a likely intended function of a while loop is to look at other loops that have similar structures [35, Section 4.4]. If two loops have similar code structures, their intended functions are likely to have similar structures as well. If we know the intended function of one loop, we may be able to adapt it to derive that of the other. Besides, many intended functions of loop control structures exhibit certain common flavors or characteristics. To capitalize on this idea, we can develop reusable patterns of while loops along with their intended functions that can be used as a valuable resource for formulating a candidate or likely intended function of a loop. As mentioned earlier, for the patterns to be useful in practice, the choice and the variety of patterns are crucial. We need to identify and accumulate a number of good patterns to cover a wide range of loops in different types of applications. A good specification 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, detailed intended function. In any pattern-based approach, it is crucial to properly document patterns [20]. Each pattern should be documented in such a way that it is easy to determine its applicability, to instantiate it in an application, and to derive a useful intended function from it. Furthermore, patterns need to be classified, organized, and presented in a pattern catalog such that they can be easily looked up and matched for. In this section we describe how we address these requirements and explain several representative patterns that we documented in our pattern catalog. Below we first describe a technique that we used to analyze while loops systematically to identify recurring patterns.

3.1. Loop Analysis

The most common use of loops is to iterate over a certain sequence of values and manipulate it, one value at a time. For example, a study indicates that 60% of loops written in C traverse arrays in some fashion—45.2% for non-string arrays and 14.3% for string arrays—and linked lists account for 13.0% [25]. A loop has a chain of steps that are performed and then repeated. There are four different types of steps or actions in the chain. The next value is obtained from the sequence being iterated over, the obtained value is manipulated, and the manipulation result is stored. A termination condition is checked to determine repetition of the steps; these steps are repeated unless (or until) a certain termination condition holds. This observation provides an excellent conceptual framework for analyzing loops systematically: *examine each of these steps or actions separately and then combine the results.* Each step or action becomes a different, orthogonal dimension for analyzing a loop. That is, a loop can be examined along the following four different analysis dimensions: (a) how it acquires the values to manipulate, (b) what manipulation it performs on the acquired values, (c) where the manipulation result is stored, and (d) when it stops its iteration. As an example, consider the following while loop taken from the sample code presented in Section 2.2.

```
1  while (i < a.length) {
2   if (a[i] > k)
3      r = r + 1;
4   i++;
5  }
```

The sequence iterated over by the loop is the elements of the array a starting from index i to the end in order, i.e., $a[i..a.length-1]$, and it is used as follows.

- Acquiring values: An index (i) is used to access elements of a ($a[i]$ in line 2) and each element is accessed in order (i++ in line 4).
- Manipulating values: if the current value is positive, compute $r+1$ (lines 2-3).
- Storing results: If the current value is positive, the manipulation result is stored in a scalar variable r (line 2). Assuming that i is an *incidental variable* used only for iterating over the sequence, there is only one non-local variable that may be updated or changed.
- Determining termination: The loop terminates when the last element is accessed (line 1), i.e., when it completes iteration over all elements of a starting from index i.

We applied this analysis framework to study a large number of while loops from several different sources including a few well-known open-source projects (see Section 5) and to identify common patterns of while loops and their intended functions. The framework is also recommended for analyzing a while loop to find a matching pattern in our pattern catalog (see Section 4). We learned that there is a wide range of possibilities along the four analysis dimensions, including several most commonly-used ones described in Figure 1. The acquisition dimension tells how the loop acquires the next value of the sequence being iterated over. As shown in [25], the sequence is frequently stored explicitly in such structures as arrays, strings, collections, streams, and files, and its elements are accessed by using indices or various forms of iterators. It is also possible to create the elements on-the-fly on a need basis instead of retrieving stored ones. The manipulation dimension determines the functionality of a loop by telling how the acquired values are manipulated or what operations are

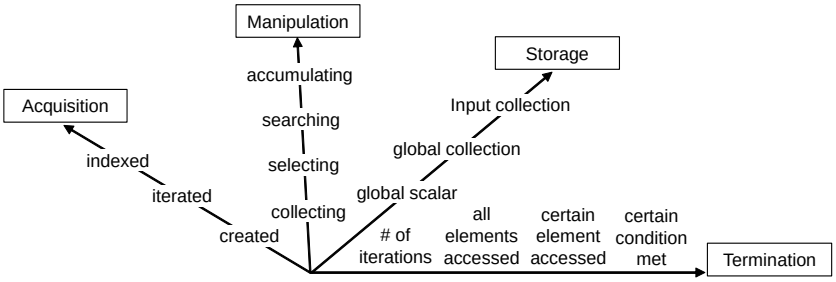

Figure 1. Dimensions of loop analysis

performed on them. It is often the most important analysis dimension, for it represents the purpose of a loop. As expected, there are numerous manipulations possible, including several common types such as accumulating, searching, counting, selecting, and collecting (see Sections 3.3–3.6). The storing dimension tells where and how the results are stored. There are also a variety of possibilities here, e.g., updating the input sequence and storing to output variables distinct from that of the input sequence. For accumulation and searching, the results are stored in scalar variables while for selecting and collecting, they are stored in vector or collection variables. The termination dimension specifies the termination condition of a loop—a condition that stops the looping. It is the opposite of the test condition that allows the loop to continue looping. A loop termination condition can differ in many ways, e.g., when all elements are accessed, when a certain element is accessed, and when a certain number of iterations has been completed. The conditions may be written in terms of indices, iterators, values of the input sequence, and others. The four dimensions allow one to analyze a loop in a modular, compositional fashion by examining each dimension separately and composing the results. For example, the above while loop is a composition of an index-based sequential acquisition, a counting manipulation, a scalar variable storage or update, and termination upon accessing all elements.

3.2. Pattern Documentation

We examined a large number of while loops from several different sources including programming textbooks, class assignments, our research projects, and open-source projects. We applied the framework and technique described above to study, group, and classify them, and from this study we identified a number of while loop patterns that are most commonly used and we documented them in a pattern catalog [4]. Some patterns are specializations or sub-patterns of other more general ones. The documented patterns are language-neutral in that they can be applied to a wide range of programming languages, from imperative, procedural languages to object-oriented languages. The patterns can be matched to while loops that iterate over different implementations of index-based collections such as arrays, strings, and sequences as well as iterator-based collections such as linked list and pointer or reference-based collection data structures commonly found in programming languages such as C, C++, and Java.

One interesting decision was documenting a pattern based on the behavior of a loop body, not its source code structure or implementation. This makes a pattern not only language neutral but also its application modular in that it can handle nested loops by first figuring out the intended function of the inner loops. As shown below, each pattern consists of two structural elements: a skeletal intended function (f_1) from which a candidate or likely intended function of a matching while loop can be derived and an intended function of the loop body (f_2).

$[f_1]$
while (E) {
  $[f_2]$
  ...
}

Table I. Main patterns documented

Name	Description
Accumulating	Combine certain elements of a collection into a single value
Unconditionally Accumulating	Accumulate all elements of a collection
Searching	Find a certain element of a collection
Selecting	Filter certain elements of a collection
Unconditionally Selecting	Select all elements of a collection
Collecting	Select and map certain elements of a collection
Unconditionally Collecting	Collect all elements of a collection

The intended function f_1 captures the behavior of the whole loop in terms of f_2 that specifies the behavior of the loop body. As mentioned earlier, the loop body is not given in skeletal code but is abstracted to an intended function so that any code segment that correctly implements the intended function can be matched to the pattern.

We documented our patterns using a format similar to that of software design patterns [20]. Each pattern has a name, purpose, description, structure, example, application, variations, and related patterns. Each pattern has a name to uniquely identify it. Then, its main purpose is described briefly, including the kind of while loops that can be matched to the pattern. The description section provides more detailed information about the pattern including its skeletal intended function. For example, it provides descriptions of main elements of skeletal intended function such as result variables and the sequence being iterated over by the loop, and explains in detail the structure of the pattern. The application section suggests a general process for applying the described pattern. It also shows a sample application of the pattern to illustrate in a step-by-step fashion how the pattern can be used. The variations and related patterns section lists variations possible for the described pattern, and some of the variations are named and catalogued separately as related patterns.

In our pattern catalog we used the manipulation dimension as the primary dimension for naming patterns, for it shows the purposes of loops—i.e., the behavior of loops. We documented seven major patterns along with numerous variations (see Table I) [4]. As can be guessed from the table, some patterns are specializations of others (see Section 3.7). The reason that we documented them as separate patterns is because they have appeared frequently in the code that we studied. In the following subsections we describe several representative patterns in detail.

3.3. Accumulating Pattern

One common use of while loops is to combine certain elements of a collection into a single value by applying various binary operators such as addition, multiplication, and concatenation. The Accumulating pattern provides a skeleton intended function for these while loops. The type of the accumulated value is often the same as that of the elements of the collection being accumulated.

3.3.1. Notation A while loop that matches the Accumulating pattern iterates over a collection of values, regardless of whether the values are read from data structures or generated on the fly. As described in Section 3.1, there are many different ways of storing and accessing the collection to be iterated over by a loop. To specify a pattern in a language-neutral and representation-independent way, we need to abstract away from these specific implementation details. Since its elements are accessed in a certain order by a loop, the collection can be viewed logically as a sequence and its elements can be denoted by specifying their positions in the sequence. The specifics of accessing elements are also abstracted to an abstract iterator. If needed, the sequence and its iterator can be defined formally as model variables [8]. Below we use the following notation to express and manipulate the collection being iterated over by a loop.

- $\langle\rangle$: an empty sequence

- $e \vdash s$: concatenation of an element e and a sequence s
- i: an abstraction of an iterator to access a sequence
- $E(i)$: an expression written in terms of the abstract iterator i. It represents an advancement of the iterator i to the next element, e.g., $i + 1$ for an index-based collection like an array and $i.next()$ for an iterator-based collection.
- $s@i$: i-th element of a sequence s, where i is an abstract iterator for s.

3.3.2. Pattern As mentioned earlier, a pattern is specified by a pair of intended functions, one for the loop body and the other for the whole loop. The Accumulating pattern is specified by referring to four different values or elements, r representing the accumulated value, s denoting the collection whose elements are accumulated, i denoting an abstract iterator for s, and $\diamond$ denoting an accumulation operator.

f_1: [r, i := $\vec{\diamond}(r, s@i..)$, anything]
while (C) {
  f_2: [r, i := $P(s@i)$? $(r \diamond s@i)$: r, $E(i)$]
}

Let's first examine the intended function of the loop body ($f2$). The loop body may change two state variables, r and i. The variable r stores the accumulated value, and i is an abstraction of the iterator to access the elements of s. The new value of r is defined by using a conditional expression of the form E_1 ? E_2 : E_3, denoting either E_2 or E_3 depending on the value of a Boolean expression E_1. The value of r is defined in terms of the following expression and operator.

- $P(x)$: a predicate defined on the elements of the sequence s. It specifies the criterion for selecting the elements to be accumulated and is a function of the signature $T \rightarrow Boolean$, where T is the element type of s. For each element x of s, it tells whether x is to be accumulated.
- $\diamond$: a binary operator of the signature $T \times T \rightarrow T$, where T is the element type of s[†]. It's an accumulation operator such as addition, multiplication, and string concatenation to combine the elements of s.

The intended function states that the current element of the sequence s (i.e., $s@i$) is accumulated in r using the accumulation operator ($\diamond$) only if it satisfies the selection criterion ($P(s@i)$). The new value of i is $E(i)$, denoting an advancement of the iterator i to the next element of s.

Let's next look at the intended function of the whole loop (f_1). It is defined by promoting the accumulation operator ($\diamond$) to the whole sequence s, denoted by $\vec{\diamond}$. The final value of r is defined in terms of the following expression and operator.

- $s@i..$: a subsequence of s starting at i, consisting of elements selected using the advancement expression $E(i)$. It is a sequence consisting of elements $s@i$, $s@E(i)$, $s@E(E(i))$, $s@E(E(E(i)))$, etc, and denotes the elements of s that are accessed by the loop. The last element is determined by the loop termination condition C; if the condition fails at the first iteration, the sequence is empty. The sequence is defined recursively.

$$s@i.. \triangleq \begin{cases} s@i \vdash s@E(i).. & \text{if } i \text{ denotes a valid position of } s \\ \langle\rangle & \text{otherwise} \end{cases}$$

  Remember that $\langle\rangle$ denotes an empty sequence and $\vdash$ denotes concatenation of an element and a sequence.
- $\vec{\diamond}$: a promotion of a binary operator $\diamond$ to a sequence. It is a function of the signature $T \times Seq(T) \rightarrow T$, where T is the argument and result type of $\diamond$ and $Seq(T)$ is a sequence

[†]The most general signature of an accumulator is $R \times T \rightarrow R$, where R is the result (accumulated value) type, allowing the accumulated value to be of different type (see Section 3.3.4).

of T. It accumulates the elements of a given sequence and a given seed value using a binary operator $\diamond$, and is defined recursively as follows.

$$\vec{\diamond}(v, \langle\rangle) \triangleq v$$
$$\vec{\diamond}(v, h \vdash t) \triangleq P(h)\ ?\ \vec{\diamond}(v \diamond h, t) : \vec{\diamond}(v, t)$$

If the given sequence is empty, it returns the seed value. If the sequence is not empty and the first element (h) satisfies the selection criterion (P), the seed value (v) and the first element (h) are combined using the accumulation operator ($\diamond$) and the function is recursively applied to the rest of the sequence. If the first element doesn't satisfy the selection criterion, it is ignored and the function is recursively applied to the rest of the sequence.

The intended function states that the final value of r is $\vec{\diamond}(r, s@i..)$, accumulation, using the $\diamond$ operator, of those elements of s at positions i, $E(i)$, $E(E(i))$, and so on that satisfy the selection criterion P.

3.3.3. Example The while loop below adds all positive elements of an array a starting at index i and stores the result to sum. In Section 4.1, it will be shown how its intended function can be derived by applying the Accumulating pattern.

```
// [sum, i := sum + Σ_{j=i..a.length−1} (a[j] > 0 ? a[j] : 0), anything]
while (i < a.length) {
  // [sum, i  := a[i] > 0 ? a[i] : 0, i + 1]
  if (a[i] > 0) {
     sum = sum + a[i];
  }
  i++;
}
```

3.3.4. Variations and Related Patterns There is a huge number of variations possible for the Accumulating pattern. Each axis of the four-dimensional loop analysis described in Section 3.1 can produce many variations, e.g., indexing vs. iterator for the acquisition dimension. Below we describe several noticeable variations that are not mentioned in the description of the four-dimensional loop analysis in Section 3.1.

- Selection: The intended function of the loop body has a general form of [r, $i := P(e)\ ?\ (r \diamond e) : r$, $E(i)$]. One possible variation is the case where the condition P is always true; there is no constraint and thus all elements are accumulated. In fact, it occurs so frequently that we defined it as a separate pattern named Unconditionally Accumulating [4]. Another possible variation is the case where the condition P is written in terms of the iterator itself, not the current element. An example is to accumulate every other element of a collection, $P(i) \triangleq i$ % 2 == 0.
- Accumulator: An accumulation operator is a binary operator such as addition, multiplication, and string concatenation. Often, its two arguments are of the same type, meaning that the accumulated value is of the same type as the element type of the sequence. In general, however, an accumulator can be of the signature $R \times T \to R$, where R is the result (accumulated value) type and T is the element type. It is also possible to have more than one accumulator, e.g., accumulating elements differently depending on certain conditions.
- Manipulation: The elements of a sequence are often transformed or manipulated prior to accumulation. To incorporate this into the pattern, the intended function of the loop body can be refined to: [r, $i := P(e)\ ?\ (r \diamond M(e)) : r$, $E(i)$]. An element e is first transformed by applying a function M: $T \to S$, that maps an element to another value, and then the accumulator $\diamond$: $R \times S \to R$ combines the transformed value. An example is to count positive values contained in an array, in which case M is a constant function that always returns 1.

- Acquisition: Beside various ways of acquiring elements described in Section 3.1, a loop can accumulate elements of more than one sequence using either a single iterator or multiple iterators. An example is to accumulate elements of two different arrays using a single iterator, e.g., [r, $i := r + a[i] + b[i]$, $i + 1$] or using two iterators, e.g., [r, i, $j := r + a[i] + b[j]$, $i + 1$, $j + 1$].
- Storage: It is possible for a loop to produce more than one accumulated value; it can have multiple result variables. An example is to sum all positive values as well as all negative values of an array; the loop body will have an intended function of the form [pos, neg, $i :=$ pos + ($a[i] > 0$? $a[i]$: 0), neg + ($a[i] < 0$? $a[i]$: 0), $i + 1$].

3.4. Searching Pattern

A while loop is frequently used to find an element in a collection, e.g., a largest value of an array. This pattern provides a skeleton intended function for those while loops that search for a particular element in a collection. The result of such a loop is typically the element found, however other results are possible, e.g., the position or index of the element found and a flag indicating whether an element is found or not. As in the Accumulating pattern, the intended function of the loop is defined by promoting the function of the loop body to a sequence.

f_1: [r, $i := \vec{\diamond}(r, s@i..)$, anything]
while (C) {
  f_2: [r, i := $P(r, s@i)$? $M(s@i)$: r, $E(i)$]
  ...
}

As specified in f_2, the loop body may change two state variables, r and i. The variable r stores the search result, and as explained previously i is an abstraction of the iterator to access the elements of the sequence s. The new value of r is defined in terms of a predicate P and a function M.

- $P(r, e)$: a predicate defined on a pair of the result value and an element of the sequence s. It specifies the search criterion for elements contained in s, and is a function of the signature $R \times T \rightarrow Boolean$, where R and T are the result type and the element type of s, respectively.
- $M(e)$: a manipulation function of the signature $T \rightarrow R$, where T is the element type and R is the result type, that transforms or maps an element to the result. Frequently it is an identity function. However, the result value doesn't have to be the element found; it can be a flag indicating the presence of an element in the sequence, which can be modeled by a constant function M that always returns true. Another common use of the manipulation function is to obtain only a certain part of a composite value, e.g., only the name of an employee.

The new value of r is the current element of s ($s@i$) transformed by M ($M(s@i)$) if the current element satisfies the search criterion ($P(r, s@i)$); otherwise, r remains the same. The new value of i is $E(i)$, denoting an advancement of the iterator i to the next element.

As in the Accumulating pattern, the intended function of the whole loop (f_1) is defined by promoting the function of the loop body to the whole sequence. Specifically, the manipulation function M is promoted to a sequence, denoted by $\vec{\diamond}$, a function of the signature $R \times Seq(T) \rightarrow R$, where R is the result type, T is the element type of the sequence s, and $Seq(T)$ is a sequence of type T. It calculates the result from a given sequence using the manipulation function M and is defined recursively.

$$\vec{\diamond}(r, \langle\rangle) \triangleq r$$
$$\vec{\diamond}(r, h \vdash t) \triangleq P(r, h) \ ? \ \vec{\diamond}(M(h), t) : \vec{\diamond}(r, t)$$

If the given sequence is empty, it returns the given result value (r). If the sequence is not empty and the first element (h) and the given result value satisfies the search criterion (P), the first element is transformed using M and the function is recursively applied to the rest of the sequence. If the first element doesn't satisfy the search criterion, it is ignored and the function is recursively applied

to the rest of the sequence. In summary, the intended function f_1 states that the final value of r is $\vec{\diamond}(r, s@i..)$, a transformed value of the elements of s at positions i, $E(i)$, $E(E(i))$, and so on that satisfy the search criterion P.

An example loop that matches the Searching pattern is shown below. It finds a maximum value of an array a starting at index i and stores it in r.

```
// [r, i := m⃗ax(r, a, i), anything]
//   where m⃗ax(r, a, i) ≜ i > a.length − 1 ? r : m⃗ax(max(r, a[i]), a, i+1)
while (i < a.length) {
  // [r, i  := a[i] > r ? a[i] : r, i + 1]
  if (a[i] > r) {
     r = a[i];
  }
  i++;
}
```

There are many variations possible for the Searching pattern. In fact, most of the variations mentioned for the Accumulating patterns are also applicable to the Searching patterns, e.g., unconditional selection, various manipulations, multiple acquisitions, and multiple results. However, most interesting variations of the Searching pattern are about the termination of the search. When searching an element in a collection, there are several different ways of terminating the search, e.g., terminating as soon as an element is found or continuing to the last element of the collection. The first case is for finding the first occurrence of a matching element and the second for finding the last occurrence. In the pattern specification, this is somewhat implicitly modeled by the sequence "$s@i..$". If needed, however, we can model the termination choice explicitly. For example, the first case can be modeled by the following definition of $\vec{\diamond}$.

$$\vec{\diamond}(e, \langle\rangle) \triangleq e$$
$$\vec{\diamond}(e, h \vdash t) \triangleq P(e, h)\ ?\ M(h) : \vec{\diamond}(e, t)$$

3.5. Selecting Pattern

This pattern provides a skeleton intended function for those while loops that select some elements of a collection and store the selected elements in the same or a different collection (see Figure 2). The element type of the result collection is the same as that of the input collection.

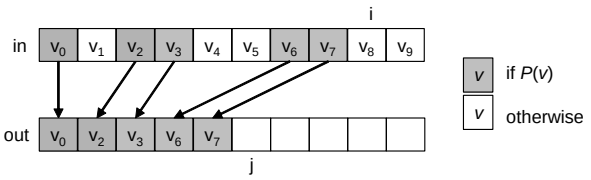

Figure 2. Selecting pattern

3.5.1. Notation The sequence notation introduced earlier is extended to specify the Selecting pattern. A collection iterated by a loop is viewed logically as a sequence, and a sequence is now modeled as a partial function from indices to elements. For example, a string sequence s consisting of two elements, say "Hello" and "World", is now viewed as a partial function from integers to strings, $\langle 0 \mapsto$ "Hello", $1 \mapsto$ "World"$\rangle$. We use the following notation to express and manipulate a sequence as a partial function.

- $s@i$: i-th element of a sequence s, where i an abstract iterator for s denoting an index; it's short for $s(i)$.

- $s@I$: a subsequence of a sequence s, consisting of elements at positions specified by an ordered index set I. It is a sequence consisting of elements projected by the index set I, e.g., $\langle 0 \mapsto 10, 1 \mapsto 20, 2 \mapsto 30\rangle @ \{0, 2\} \equiv \langle 0 \mapsto 10, 2 \mapsto 30\rangle$
- dom: domain of a sequence, e.g., $dom\ \langle 0 \mapsto 10, 1 \mapsto 20\rangle \equiv \{0, 1\}$. The result is an ordered set.
- ran: range of a sequence, e.g., $ran\ \langle 0 \mapsto 10, 1 \mapsto 20\rangle \equiv \{10, 20\}$. The result is an ordered bag.
- $\uplus$: function overriding. The expression $f_1 \uplus f_2$ maps everything in the domain of f_2 the same value as f_2 does, and everything else in the domain of f_1 to the same value as f_1 does, e.g., $\langle 0 \mapsto 10, 1 \mapsto 20\rangle \uplus \langle 1 \mapsto 30, 2 \mapsto 40\rangle \equiv \langle 0 \mapsto 10, 1 \mapsto 30, 2 \mapsto 40\rangle$. If the domains of two functions are disjoint, it is the union of the two functions.

3.5.2. Pattern The pattern is specified by referring to the input and the result collections (in and out) along with their iterators (i and j) and the criterion for selecting elements (P) (see below). The iterators are used to access and store the elements of collections. The variable in denotes the collection whose elements are to be selected. Since its elements are accessed in a certain order in a loop, it is viewed logically as a sequence, and its elements are denoted by their positions in the sequence. For this, an abstract variable i—an abstraction of the iterator to access the elements of the collection—is introduced, and the notation $in@i$ is used to denote the i-th element of the sequence in. Similarly variables out and j are used to denote the result sequence and its iterator, respectively.

f_1: [$out@D$, i, j := R, anything, anything]
  where D and R are domain and range of $\vec{\diamond}(in, out, i, j, \langle\rangle)$
while (C) {
  f_2: [$out@j$, i, j := $P(in@i)$? $in@i$: $out@j$, $E_1(i)$, $P(in@i)$? $E_2(j)$: j]
  ...
}

The intended function of the loop body (f_2) states that the loop body may change three state variables, out, i and j. The variable out contains the selected elements, and i and j are abstractions of the iterators to access the elements of in and out. P is a predicate defined on the elements of the sequence in. It's a function of the signature $T \rightarrow Boolean$, where T is the element type of in, and specifies the selection criterion. If $P(x)$ is true for an element x of in, x should be selected. The new value of $out@j$ is the current element of in (i.e., $in@i$) if the current element satisfies the selection criterion ($P(in@i)$); otherwise, it's the same as the old value. The iterators i and j advance to the next elements, however, for j only if $s@i$ is selected. Operationally, the intended function states that if the element in in at position i satisfies the condition P, it will be stored in out at position j; otherwise, the element of out at position j remains the same.

Now let's examine the intended function of the whole loop (f_1). The loop selects the elements of in that satisfy the selection condition P and stores the selected elements in out. The intended function f_1 specifies this behavior by promoting the selection and storing of individual elements to the whole sequences, as denoted by $\vec{\diamond}$. Remember that the notation $out@D$ denotes a subsequence of out indexed by an ordered index set D, where D is the domain of $\vec{\diamond}$. The function $\vec{\diamond}$ determines the elements (of in) to be selected along with their new indices (in out). It is defined by promoting the intended function of the loop body specified at the element level to a sequence and is defined recursively as follows.

$$\vec{\diamond}(in, out, i, j, r) \triangleq$$
$$\begin{array}{ll} r & \text{if } \neg C(in, out, i, j) \\ \vec{\diamond}(in, out, E_1(i), E_2(j), r \uplus \langle j \mapsto in@i\rangle) & \text{if } C(in, out, i, j) \wedge P(in@i) \\ \vec{\diamond}(in, out, E_1(i), j, r) & \text{otherwise} \end{array}$$

The last argument (r) is an accumulator storing the index-value pairs of the selected elements. The condition C is the loop termination condition and may be written in terms of in, out, i and j. The three cases represent (1) when all iterations are completed, (2) when the current element is selected

as it satisfies the selection criterion, and (3) when the current element is not selected as it doesn't satisfy the selection criterion. The $\uplus$ symbol denotes function overloading. With this definition, $\vec{\diamond}(in, out, i, j, \langle\rangle)$ denotes the selected elements as a partial function from indices to values.

3.5.3. Example The while loop below copies all positive elements of an array a starting at index i to b starting at index j.

/* [b[j..j+n−1], i, j := $\vec{\diamond}$[i..a.length−1], anything, anything]
 * where n is the number of positive values in array a starting at index i and $\vec{\diamond}$ is defined below.
 * $\vec{\diamond}(\langle\rangle) \triangleq \langle\rangle$
 * $\vec{\diamond}(h \vdash t) \triangleq h > 0 \ ?\ h \vdash \vec{\diamond}(t) : \vec{\diamond}(t)$ */
while (i < a.length) {
 // [b[j], i, j := a[i] > 0 ? a[i] : b[j], i + 1, a[i] > 0 ? j + 1 : j]
 if (a[i] > 0) {
  b[j] = a[i];
  j++;
 }
 i++;
}

3.5.4. Variations and Related Patterns Like previous two patterns there are many variations possible for the Selecting pattern. Most of the variations mentioned for the Accumulating pattern are also applicable to this pattern, including various manipulations, multiple acquisitions and multiple results. Below we describe several noticeable variations, specific to the Selecting pattern.

- Selection: The intended function of the Selecting pattern has a general form of $[out@j, i, j := P(in@i)\ ?\ in@i : out@j, E_1(i), E_2(j, in@i)]$, and one variation is the case where the selection condition P is always true; that is, all elements are selected. Since it occurs so frequently we documented it as a separate pattern named Unconditionally Selecting [4].
- Transformation: The selected elements may be transformed before they are collected. In fact, it is so common that it was documented and cataloged as a separate pattern named Collecting pattern (see Section 3.6). The Selecting pattern is a specialization of the Collecting pattern where the transformation is an identity function.
- Storage: Instead of storing the selected elements to another collection, it is possible to store them to the input collection, e.g., shifting elements $[a[i-1], i := a[i], i+1]$.

3.6. Collecting Pattern

A while loop is often used to collect certain elements of a collection. It picks elements that satisfy a certain condition, transform them, and stores the results in the same or a different collection. The Collecting pattern captures this use of while loops. It is a generalization of the Selecting pattern (see Section 3.5), and the element type of the result collection may be different from that of the input collection.

3.6.1. Pattern As in the Selecting pattern, the intended function of the loop is defined by referring to the input and the result collections (in and out) along with their iterators (i and j), the element selection criterion (P), and the function to transform the selected elements (M). In fact, the specification of this pattern is almost identical to that of the Selecting pattern, and the only difference is the introduction of a transformation function denoted by M.

f_1: [$out@D$, i, j := R, anything, anything]
 where D and R are domain and range of $\vec{\diamond}(in, out, i, j, \langle\rangle)$
while (C) {
 f_2: [$out@j$, i, j := $P(in@i)\ ?\ M(in@i) : out@j$, $E_(i)$, $P(in@i)\ ?\ E_2(j) : j$]
 ...

}

The intended function of the loop body (f_2) states that the loop body may change three state variables, out, i and j. The variable out contains the collected elements, and i and j are abstractions of the iterators to access the elements of in and out. P is a predicate defined on the elements of the sequence in. It's a function of the signature $T \rightarrow Boolean$, where T is the element type of in, and specifies the selection criterion. If $P(x)$ is true for an element x of in, x should be collected. M is a function defined on the elements of the sequence in with a signature $T \rightarrow R$, where T and R are the element type of in and out, respectively, It maps or transforms the selected elements to possibly different values. The new value of $out@j$ is the current element of in (i.e., $in@i$) transformed using M if the current element satisfies the selection criterion ($P(in@i)$); otherwise, it's the same as the old value. The iterators i and j advance to the next elements, however, for j only if $s@i$ is collected. Operationally, the intended function states that if the element in in at position i satisfies the condition P, it will be stored in out at position j after transformed using M; otherwise, the element of out at position j remains the same.

As expected, the intended function of the whole loop (f_1) is defined by promoting the selection, transformation and storing of individual elements to the whole sequences, as denoted by $\vec{\diamond}$. The function $\vec{\diamond}$ gives the transformed values of the elements (of in) to be collected along with their new indices (in out), and its definition is identical to that of the Selecting pattern except for the use of a transformation function M.

$$\vec{\diamond}(in, out, i, j, r) \triangleq$$
$$\begin{array}{ll} r & \text{if } \neg C(in, out, i, j) \\ \vec{\diamond}(in, out, E_1(i), E_2(j), r \uplus \langle j \mapsto M(in@i)\rangle) & \text{if } C(in, out, i, j) \wedge P(in@i) \\ \vec{\diamond}(in, out, E_1(i), j, r) & \text{otherwise} \end{array}$$

With the above definition, $\vec{\diamond}(in, out, i, j, \langle\rangle)$ denotes the collected elements as a partial function from indices to values, whose range (R) becomes the new value of $out@D$.

3.6.2. Example The while loop below collects all positive elements of an array a starting at index i by multiplying 2 to them and storing the results in an array b starting at index j. In Section 4.2 we will show how the intended function of a similar loop can be derived by applying the Collecting pattern.

```
/* [b[j..j+n−1], i, j := ⋄⃗[i..a.length−1], anything, anything]
 * where n is the number of positive values in array a starting at index i and ⋄⃗ is defined below.
 *  ⋄⃗(⟨⟩) ≜ ⟨⟩
 *  ⋄⃗(h ⊢ t) ≜ h > 0 ? h * 2 ⊢ ⋄⃗(t) : ⋄⃗(t)  */
while (i < a.length) {
 // [b[j], i, j := a[i] > 0 ?  a[i] *2 : b[j], i + 1,  a[i] > 0 ? j + 1  :  j]
 if (a[i] > 0) {
    b[j] = a[i] * 2;
    j++;
 }
 i++;
}
```

3.6.3. Variations and Related Patterns All the variations of the Selecting pattern are also applicable to the Collecting pattern, for the Selecting pattern is a specialization of the Collection pattern in which the transformation function is an identity function. As in the Selecting pattern, if the collecting condition is always true, all elements are collected, and this is documented and cataloged as a separate pattern named Unconditionally Collecting. There is a wide range of transformations possible, e.g., collecting indices of the elements not the elements themselves, and due to the transformation, many interesting variations are possible along the storage dimension. For example, a loop may have more than one result collection, e.g., element-wise sum and product of two

arrays which can be accomplished by a loop body with an intended function $[sum[i], prod[i], i := a[i] + b[i], a[i] * b[i], i + 1]$.

3.7. *Discussion*

The most common use of loop control structures is to iterate over a certain sequence of values and manipulate the values of the sequence, regardless of whether the values are retrieved from data structures or created on-the-fly. A loop pattern is defined in terms of the manipulation of individual values specified by the intended function of the loop body. In particular, the manipulation of individual values is promoted to the whole sequence to specify the intended function of the whole loop. Therefore, depending on how we define the manipulation of individual values, we can have a wide variety of patterns possibly at many different levels of abstraction. At the highest level of abstraction, the intended function of the loop body may be written as $[r, i := F(\vec{v}), E(\vec{v})]$, where F and E calculate new values of r and i, respectively, in terms of the initial values of variables $\vec{v}$ that may include r, i, and of course the sequence being iterated over, and the intended function of the whole loop can be defined by promoting or extending F to the whole sequence. For example, if the intended function of a loop body is $[r := r + (a[i] * 2)]$, F is defined as $F(r, a, i) \triangleq r + (a[i] * 2)$. At the next level of abstraction, the function F can be further decomposed into $S(M(\vec{v}), \vec{v})$, where M is an abstraction of the individual value manipulation and S is a storage function. This level of abstraction corresponds to the way we examine a loop along the four analysis dimensions. For the same intended function $[r := r + (a[i] * 2)]$, F is now refined to $S(r, M(a, i))$, where $M(a, i) \triangleq a[i] * 2$ and $S(r, x) \triangleq r + x$. Each of M, F, and E can be further decomposed or refined, say to introduce a condition to model a conditional manipulation, storage, or advancement as done in some of our patterns.

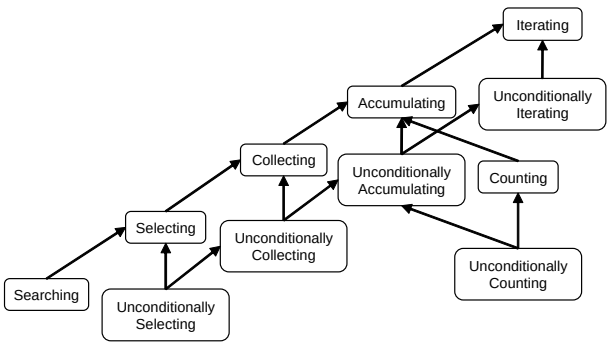

Figure 3. Pattern hierarchy

There is one nice consequence of decomposing value manipulations and defining patterns by promoting individual value manipulations to sequences. Patterns can be classified into a pattern hierarchy (see Figure 3). There exists at the root of the hierarchy a pattern whose loop body has an intended function $[r, i := F(\vec{v}), E(\vec{v})]$. It's sort of a universal pattern applicable to any loop that iterates over a sequence of values, but it's less useful in practice because it's so abstract; it doesn't provide much help in understanding a loop or guiding derivation of a detailed, likely intended function of the loop. A generalized pattern is applicable to a wide range of loops, but its specification is more abstract and thus provides less help in deriving a detailed intended function from its application. A specialized pattern, on the other hand, is more specific with limited applicability but provides more help in deriving a detailed intended function. The pattern hierarchy is extensible in that one can easily define and add a new pattern by refining or specializing the value manipulation function of an existing pattern. For example, we can introduce a new sub-pattern of Accumulating, named Counting, to count the number of elements of a collection that meet a certain condition (see Figure 3). For this, the function of Accumulating, $P(s@i)\ ?\ (r \diamond s@i)\ :\ r$, is refined

to $P(s@i)\ ?\ (r+1)\ :\ r$[‡]. The pattern hierarchy can also be used to find matching patterns for a loop by starting from more general patterns moving down to more specific ones.

Our patterns are compositional in two different senses. A pattern can be decomposed along the four, orthogonal dimensions of the loop analysis: value acquisition, value manipulation, loop termination, and result storage (see Section 3.1). Even though our patterns are named along the value manipulation dimension, each dimension contributes to the definition of a pattern and, in fact, produces new patterns or variations, typically more specific ones, e.g., Index-based Accumulating. As a consequence, a new pattern can be assembled by selecting an appropriate combination of values from the four analysis dimensions, one from each dimension (see Section 4.1). A loop can change more than one non-local variable, and our patterns can be used to derive an intended function of such a loop. An appropriate pattern is applied for each result variable to determine its final value, and all the variables along with their final values are listed together in an intended function to come up with an intended function of the whole loop (see Section 4.2 for an example).

4. APPLICATION OF PATTERNS

In this section we first suggest a general process for applying all documented patterns and their variations. We then apply two of our patterns to sample code. The following four steps are recommended for applying a pattern to derive an intended function of a while loop.

1. Formulate an intended function of the loop body.
2. Find a matching pattern from the pattern catalog.
3. Unify intended functions of the code and the pattern.
4. Instantiate the intended function of the pattern.

The first step is to formulate and specify the behavior of the loop body, for a pattern is specified in terms of the intended function of the loop body, not its code structure. If the code of the loop body doesn't contain any nested loops, its code function may be systematically calculated using techniques like trace tables [35, 36]. Essentially, one will need to identify and list all the state variables that are mutated by the code and specify their new values typically in terms of their old values. If the loop body contains other loops, however, the intended functions of the nested loops can be found first by applying the patterns from the pattern catalog. In any case, the intended function or code function of the loop body should document all the side effects produced by the loop body, i.e., state changes caused by a single iteration of the loop. Note that it is possible for a loop to have more than one input collection or output variable (see below).

Once the behavior of the loop body is formulated and specified in an intended function, the next step is to match the loop to one of the patterns documented in the catalog. For this, it is suggested to examine the loop along the four analysis dimensions described in Section 3.1: (a) how it acquires the values to manipulate, (b) what operation or manipulation it performs on the acquired values, (c) where and how the manipulated value is stored, and (d) when it terminates the iteration. Most of the analysis, especially acquisition, manipulation, and storage are likely to have been performed already and documented in the intended function of the loop body. The loop body will have an intended function of the following general form:

$$[s_1, s_2, \cdots, s_n := M_1(e, s_1), M_2(e, s_2), \ldots, M_n(e, s_n)]$$

where s_i is a state variable whose value may be changed in the loop body, e is the current element of the collection being iterated over, M_i is a manipulation function defining the new value of s_i usually in terms of its old value and the current element of the collection. The state variable s_i is either a result variable or an iterator, and the current element e is typically given in terms of an iterator. If a loop has more than one result variable, one needs to find a pattern and apply it for

[‡]It is also possible to refine the function of Searching, $P(r, s@i)\ ?\ M(s@i) : r$, to $P(s@i)\ ?\ (r+1)\ :\ r$.

each result variable; it is also possible for a loop to have more than one input collection. To find a matching pattern, compare the manipulation function, M_i, with those of the patterns in the catalog. For example, M_i can be matched to the Accumulating pattern if it has the form $P(e)\ ?\ e \diamond s_i : s_i$, where s_i is a result variable, P is a predicate defined on the elements of an input collection, $\diamond$ is a binary (accumulation) function defined on a tuple of the result and an element of the input collection (see Section 3.3 for the Accumulating pattern).

Once a matching pattern is found, the next step is to define a mapping or correspondence between variables, symbols, and expressions appearing in the intended functions of the loop body of the code and the matched pattern. This mapping will allow one to derive an intended function of the code from the skeletal intended function given by the pattern.

The last step is to derive an intended function of the code by instantiating the skeletal intended function of the pattern. For this, one needs to replace variables, symbols, and expressions appearing in the skeletal intended function with the corresponding ones of the code, given by the binding defined in the previous step.

4.1. Accumulating Pattern

In this subsection we illustrate in detail an application of the Accumulating pattern using the example loop shown in Section 3.3, which is copied below.

```
while (i < a.length) {
  if (a[i] > 0) {
     sum = sum + a[i];
  }
  i++;
}
```

We first formulate the intended function of the loop body. The code function of the loop body can be written straightforwardly; $a[i]$ is added to sum only if it is positive, and i is always incremented by 1. Thus, its code function is: $[sum, i := a[i] > 0\ ?\ sum + a[i] : sum, i + 1]$.

We next find a matching pattern. The loop body of the Accumulating pattern has an intended function of the form $[r, i := P(s@i)\ ?\ s@i \diamond r : r, E(i)]$, where r is a result variable, i is an iterator, P is a predicate defined on the elements of an input collection, $\diamond$ is a binary (accumulation) function defined on a tuple of the result and an element of the input collection (see Section 3.3 for the Accumulating pattern). The structures of both functions are identical. The intended function of the loop body matches that of the Accumulating pattern with the binding $\{r \mapsto sum, i \mapsto i, e \mapsto a[i], P(e) \mapsto e > 0, e \diamond r \mapsto r + e, E(i) \mapsto i + 1\}$. It is also easy to see that the loop has the following characteristics; decision trees such as the ones shown in Figure 4 can be useful in identifying loop characteristics.

- Acquisition: index-based (i, $a[i]$) and sequential ($i + 1$)
- Manipulation: addition ($+$)
- Storage: scalar variable update (sum)
- Termination: when all elements are accessed ($i < a.length$)

We unify intended functions of the code and the matching pattern. We map terms such as variables, symbols and expressions from the matching pattern to those of code, and the result is summarized in Table II.

Finally, we can now instantiate the skeletal intended function of the pattern using the binding defined in the previous step (see Table II).

$$[r, i := (r, s@i..), anything] \equiv [sum, i := \vec{\diamond}(sum, a[i..a.length - 1]), anything]$$

where is $\vec{\diamond}$ also instantiated as follows.

$$\vec{\diamond}(v, \langle\rangle) \triangleq v$$
$$\vec{\diamond}(v, h \vdash t) \triangleq v > 0\ ?\ \vec{\diamond}(v + h, t) : \vec{\diamond}(v, t)$$

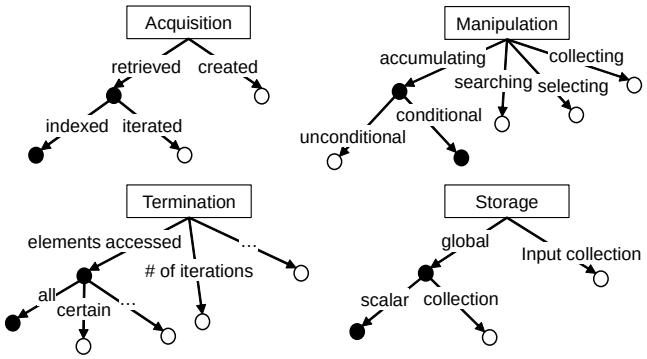

Figure 4. Decision trees for analyzing a loop

Table II. Mapping of terms

Pattern		Code	
Intended function	Term	Term	Intended function
	s	a	
	r	sum	
$[r, i := P(s@i)\ ?$	i	i	$[sum, i := a[i] > 0\ ?$
$(r \diamond s@i) : r, E(i)]$	$P(x)$	$x > 0$	$sum + a[i] : sum, i + 1]$
	$x@i$	$x[i]$	
$[r, i := \vec{\diamond}(r, s@i..), anything]$	$x \diamond y$	$x + y$	
	$E(x)$	$x + 1$	
	$x@i..$	$x[i..a.length - 1]$	

Note that $\vec{\diamond}$ denotes the sum of all positive elements of the given array plus the given value, and thus it can be rewritten using a more familiar mathematical notation: $\vec{\diamond}(sum, a[i..a.length]) \equiv sum + \Sigma_{j=i..a.length-1}(a[j] > 0\ ?\ a[j] : 0)$. Therefore, the derived intended function can be rewritten as:

$$[r, i := sum + \Sigma_{j=i..a.length-1}(a[j] > 0\ ?\ a[j] : 0), anything]$$

which matches the intention of the loop, i.e., calculating the sum of all positive numbers stored in the array a starting at index i.

4.2. Collecting Pattern

In this subsection we will analyze a code snippet taken from a Battleship game server written in Java. Battleship is a guessing game played by two players on grids, usually 10×10, of squares (see Figure 5). Each player has a fleet of ships and each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically.

The code shown in Listing 2 is excerpted from a method that processes a ships deployment message sent by a Battleship client, requesting to place a player's ships on the opponent's board. The body of a deployment message is a string of the form $n_1, s_1, x_1, y_1, b_1, ..., n_m, s_m, x_m, y_m, b_m$, where n_i is the name of a ship, s_i is its size, x_i and y_i are the coordinate of the starting square, b_i is its direction, true for horizontal and false for vertical. An example deployment message is: "Aircraft carrier, 5, 10, 1, false, Battleship, 4, 2, 1, true, Frigate, 3, 2, 3, false, Submarine, 3, 3, 9, true, Minesweeper, 2, 4, 10, true". The loop takes the body of a deployment message, given as a string tokenizer named *tokens* of type StringTokenizer, and processes it by placing ships at specified squares on a Battleship board named *board* (see Figure 6).

Let's analyze the loop in isolation and derive its intended function. The loop has an input variable, $tokens$, and two output variables, $board$ and $noError$. The variable $tokens$ is an input collection,

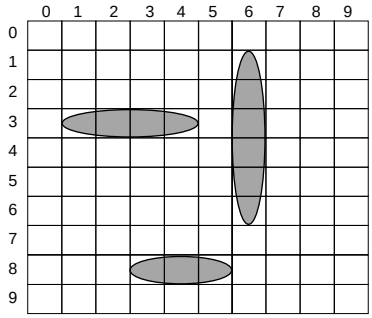

Figure 5. Battleship board

Listing 2. Code from a Battleship game server

```
1  StringTokenizer tokens = new StringTokenizer(msgBody, ‘‘,’’);
2  boolean noError = true;
3  while (noError && tokens.hasMoreTokens()) {
4     try {
5        String name = tokens.nextToken();
6        int size = Integer.parseInt(tokens.nextToken());
7        int x = Integer.parseInt(tokens.nextToken());
8        int y = Integer.parseInt(tokens.nextToken());
9        boolean dir = Boolean.parseBoolean(tokens.nextToken());
10       Battleship ship = new Battleship(name, size);
11       noError = board.placeShip(ship, x, y, dir)
12    } catch (Exception e) {
13       noError = false;
14    }
15 }
```

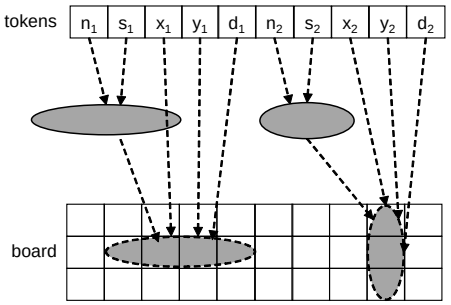

Figure 6. Behavior of the loop

however, it can also be regarded as an output variable if one cares about its final value; its state can be changed by a built-in iterator ($nextToken$). We will abstract from this specific implementation detail and use a pseudo variable i to denote its iterator as we did in our patterns documentation. To find a matching pattern, we first need to formulate the intended function of the loop body. Note that the loop body make a call to the $placeShip(ship, x, y, dir)$ method defined in the Board class. We need to know its behavior, ideally documented in an intended function. Let’s assume its behavior is

specified as follows.

$$[result, this := placeable(this, ship, x, y, dir), \\ placeable(this, ship, x, y, dir)\ ?\ this \oplus (ship, dir) : this]$$

where pseudo variables $result$ and $this$ represents the return value and the receiver, and $placeable(b, s, x, y, d)$ is a predicate telling whether a ship s can be placed at a position (x, y) of a board b horizontally or vertically (d). A ship can be placed on a board if it doesn't overlap with other ships. The $\oplus$ operator models placement of a ship on a board; the result is the same as the given board except that the specified squares are now occupied by the given ship. We now can formulate the intended function of the loop body consisting of a `try-catch` statement. We will consider the `try` clause first. All the variables except for $noError$ and $board$ are local variables and invisible in the final state, and thus they shouldn't appear in the intended function of the `try` clause shown below.

$$[noError, board, i := spl,\ spl\ ?\ board \oplus (s, x, y, d) : board,\ i + 5]$$

where $spl \triangleq placeable(board, s, x, y, d)$, $s \triangleq$ new Battleship(n, l), $n \triangleq tokens@i$, $l \triangleq tokens@(i + 1)$, $x \triangleq tokens@(i + 2)^\tau$, $y \triangleq tokens@(i + 3)^\tau$, and $d \triangleq tokens@(i + 4)^\tau$. For string v, we use the notation v^τ to model parsing v to the value of an appropriate type (int or boolean). The intended function essentially states that tokens from $tokens$ are transformed to appropriate values (int, boolean, and Battleship) and the results are stored in $board$. The intended function of the `catch` clause is $[noError := false]$. We combine both functions to come up with an intended function of the loop body.

$$\begin{aligned} &[tokensOk \wedge placeable(this, ship, x, y, dir) \rightarrow \\ &\quad noError, board, i := true, board \oplus (s, x, y, d),\ i + 5 \\ &\mid otherwise \rightarrow result, i := false, i + \delta] \end{aligned} \tag{1}$$

where $tokensOk$ is a predicate indicating the existence of four more tokens and their well-formedness (no parsing error). δ is an offset in the range of 0 and 4; it is the offset of the first token that is not well-formed, the offset of the last token if there exists less than 4 tokens in $tokens$, or 4 otherwise. Note that we use a *conditional concurrent assignment*, a concurrent assignment that may have an optional condition or guard followed by an $\rightarrow$ symbol [35, 36]. It specifies a partial function that is defined only when the condition holds.

Before we match the above intended function to a pattern, let's examine the loop along the four different analysis dimensions. We can easily see the following characteristics.

- Acquisition: built-in iterator of StringTokenizer ($nextToken()$)
- Manipulation: transformation to battleships for $board$ and to true/false for $noError$
- Storing: scalar ($noError$) and collection ($board$)
- Termination: when all elements are accessed ($hasMoreTokens()$) or upon an error

As shown above, the code mutates two state variables, $board$ and $result$, and each can be matched to a different pattern to find its final value. Let's first consider the main state variable $board$. From the intended function 1 above, we can extract those parts that are concerned with the side-effect on $board$.

$$[board := tokensOk \wedge placeable(board, s, x, y, d)\ ?\ board \oplus (s, x, y, d) : board]$$

Structurally it can be matched to both the Accumulating and the Collecting patterns. It depends on one's view of a board, a single entity (accumulation) or an aggregation (collection) of ships. Since we have already shown an application of the Accumulating pattern in the previous subsection, we will match it to the Collecting pattern. In fact, this is a sensible decision, for we can rewrite the intended function as $[board@(x, y) := tokensOk...\ ?\ (s, d) : board@(x, y)]$ using an indexing

notation; a board is viewed abstractly as a map from x-y coordinates to ship-direction pairs. When it is convenient, we will use this indexing notation and the map view below. The intended function of the Collecting pattern is $[out@D := R]$, where D and R are the domain and the range of $\vec{\diamond}(in, out, i, j, \langle\rangle)$, and $\vec{\diamond}$ is defined as follows (see Section 3.6).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\vec{\diamond}(in, out, i, j, r) \triangleq} \\
\quad r & \text{if } \neg C(in, out, i, j) \\
\quad \vec{\diamond}(in, out, E_1(i), E_2(j), r \uplus \langle j \mapsto M(in@i)\rangle) & \text{if } C(in, out, i, j) \wedge P(in@i) \\
\quad \vec{\diamond}(in, out, E_1(i), j, r) & \text{otherwise}
\end{array}
$$

Remember that i and j are abstract iterators for in and out, respectively. The pattern's intended function can be instantiated to $[board@D := R]$, where D and R are the domain and the range of $\vec{\diamond}(tokens, board, i, \langle\rangle)$ and $\vec{\diamond}$ is defined below. Note that the iterator j is dropped because it is unified to (x, y) calculated on the fly.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\vec{\diamond}(tokens, board, i, r) \triangleq} \\
\quad r & \text{if } \neg(tokensOk \wedge placeable(board, s, x, y, d)) \\
\quad \vec{\diamond}(tokens, board, i + 5, r \uplus \langle (x, y) \mapsto (s, d)\rangle) & \text{otherwise}
\end{array}
$$

where s, x, y, d are defined as before; they are parsed from $tokens$ using the iterator i. Note that recursion terminates if current tokens are not well formed or the parsed ship can't be placed on the board.

We next consider the $result$ variable, whose final value can be specified as $[result := tokensOk \wedge placeable(this, s, x, y, d)]$, extracted from the intended function 1 above. The intended function can be matched to the Searching and the Accumulating patterns, e.g., searching for an erroneous situation or accumulating boolean values. In either case, the pattern's intended function can be straightforwardly instantiated to $[result := \vec{\diamond}(tokens, board, i)]$, where $\vec{\diamond}$ is defined recursively as follows; we overload the $\vec{\diamond}$ symbol for calculating values of $board$ and $result$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\vec{\diamond}(tokens, board, i) \triangleq} \\
\quad true & \text{if } i \text{ is invalid (i.e., no more token)} \\
\quad false & \text{if } \neg(tokensOk \wedge placeable(board, s, x, y, d)) \\
\quad \vec{\diamond}(tokens, board \oplus (s, x, y, d), i + 5) & \text{otherwise}
\end{array}
$$

The last step is to combine the two intended functions, and we have $[board@D, result, i := R, \vec{\diamond}(tokens, board, i), anything]$, where D and R are the domain and the range of $\vec{\diamond}(tokens, board, i, \langle\rangle)$. If one cares about the final value of $tokens$, then the final value of i can be defined precisely by overloading $\vec{\diamond}$ as done for $result$.

5. EVALUATION

We performed a case study to evaluate our patterns, in particular, to determine their applicability in real applications. We applied our patterns to source code of several open source projects available from the Apache Software Foundation (http://www.apache.org). We picked several target applications for our study to address the great diversity in software applications, e.g., systems programming, business applications, scientific applications, and Web software. We restricted the implementation languages to Java and C, two of the programming languages that we are most familiar with and that are also the most popular in practice. Listed below are the target applications picked up for our case study.

- Chukwa 0.5: An open source data collection system for monitoring large distributed systems including a toolkit for for displaying, monitoring and analyzing the collected data [16]. It is written in Java and JavaScript.
- Commons Math 3.3: A library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language [17].

- HTTP Server 2.0.65: A popular, open-source HTTP server written in C [13].
- Jmeter 2.11: An application designed to test and measure performance of Web applications [14]. It is written in Java.
- Syncope 1.2.0-M1: a framework and system for managing digital identities in database applications and enterprise environments, implemented in Java EE [15].

We found total 2180 while loops in the source code of these applications, and among these we pick 126 loops for our study (see Table III). From each application we first selected randomly source code files containing while loops and then picked one arbitrary loop from each selected file.

Table III. Number of loops

	Sources		Samples
	Loops	Files	Loops/Files
Chukwa	173	60	24
HTTP	1300	250	13
Jmeter	335	158	25
Math	348	164	55
Syncope	24	16	9
Total	2180	648	126

The initial plan for our study was to derive the intended functions of all the selected loops by following the step-by-step processes described in the applicable patterns, working one loop at a time. However, we soon learned that many loops have similar flavors or structures, and the processes of deriving their intended functions are almost identical. Thus, for groups of similar loops we applied our patterns only to one or two representative loops, and for the rest of loops we just identified unless there are any interesting aspects on the pattern applications.

Table IV summarizes the coverage of our patterns measured in the number of loops that were successfully matched to our patterns and thus whose intended functions were derived, or likely to be derivable, from the patterns. The column labeled “Mul” shows the number of loops that were matched to more than one pattern, i.e., loops that have more than one primary output variable, and the “Not” column shows loops that couldn’t be matched to any of our patterns (see below for details). The result is very promising in that 96% of loops were matched to our patterns, meaning that their intended functions were, or could be, derived using the patterns. It’s also interesting to learn that the distributions of matching patterns vary among applications (see Figure 7), but on average the Collecting pattern occurs most commonly at 36%.

Table IV. Statistics of matching patterns

	Acc (%)	Sea (%)	Sel (%)	Col (%)	Mul (%)	Not (%)	Total
Chukwa	4 (17)	1 (4)	6 (25)	9 (38)	4 (17)	0 (0)	24
HTTP	4 (31)	2 (15)	2 (15)	5 (38)	0 (0)	0 (0)	13
Jmeter	3 (12)	1 (4)	11 (44)	3 (12)	4 (16)	3 (12)	25
Math	11 (20)	18 (33)	8 (15)	13 (24)	3 (5)	2 (4)	55
Syncope	2 (22)	1 (11)	0 (0)	6 (67)	0 (0)	0 (0)	9
Total	24 (19)	23 (18)	27 (21)	36 (29)	11 (9)	5 (4)	126
					121 (96)		

Below we describe some of observations, findings, and lessons learned from our case study along with other interesting topics of discussion. There are several loops that didn’t match any of our patterns. These are loops mainly written for non-functional behavior. In Jmeter, for

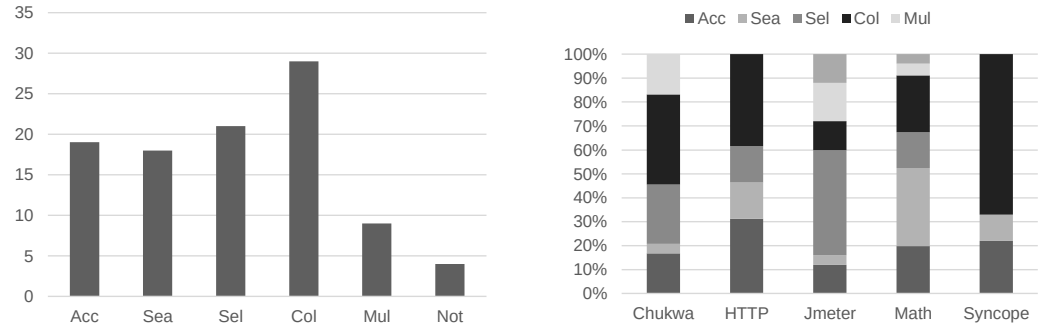

Figure 7. Percentage of matching patterns: overall (left) and individual applications (right)

example, we found several loops written for clock, timing, and concurrency using methods like System.currentTimeMillis() and Thread.sleep(). They don't match any of our patterns, for our patterns are for functional aspects of sequential programs. The two loops from the Commons Math package are concerned about GUI operations calling GUI methods such as repaint(). In theory it's possible to model them as state changes, but in practice there is no benefit of doing so, for there is a better way of modeling user interfaces.

As a study has shown, the most common use of loops is to iterate over a certain sequence of values, stored explicitly in data structures such as arrays [25]. We learned that such loops are relatively easy to analyze in order to find matching patterns. More difficult ones are those that generate values on the fly. In Commons Math, loops are mostly used for performing mathematical calculations involving all sorts of numerical operations. A significant number of loops iterate on numbers determined on the fly, not over a stored sequence of numbers, and often it's not straightforward to figure out the sequence of numbers being iterated over. However, once the imaginary sequences of numbers are identified and defined correctly, the applications of patterns are often straightforward. For example, the following loop from the Commons Math package takes two numbers a and b, and repeats the loop body an indefinite number of times.

```
while (a != b) {
  final int delta = a − b;
  b = Math.min(a, b);
  a = Math.abs(delta);
  a ≫= Integer.numberOfTrailingZeros(a);
}
```

The next values of a and b are determined on the fly, i.e., $min(a, b)$ and $|a - b|^{\gg}$, where $x^{\gg}$ denotes a right shifting of x by the number of trailing zero bits. Abstractly, the loop can be thought of taking two sequence of numbers, say $\vec{a}$ and $\vec{b}$, determined by the initial values of a and b, and iterate over them. If a and b are initially 10 and 7, then $\vec{a}$ and $\vec{b}$ will be $\langle 10, 3, 1, 1\rangle$ and $\langle 7, 7, 3, 1\rangle$. And the final values of both a and b will be 1, for the loop searches for a pair-wise equivalent value, which is always the last element in the sequence.

There were cases that we have to change loop code a bit to apply our patterns. In Jmeter, for example, there are lots of while loops that call test oracle methods such as assertEquals() that may throw an exception and thus terminate a loop abruptly. Operationally they are similar to loops that contain an exit type of control statements such as *break* and *return* statements. To match such a while loop to one of our patterns, we first had to rewrite the loop code slightly. As an example, consider the following while loop take from Jmeter.

```
while ((sampler = controller.next()) != null) {
  assertEquals(order[counter++], sampler.getName());
}
```

It can be rewritten to the following by introducing a flag, say $testOk$, indicating a test success or failure.

```
while (testOk && (sampler = controller.next()) != null) {
  try {
    assertEquals(order[counter++], sampler.getName());
  } catch (AssertionError e) {
    testOk = false;
  }
}
```

Once it's rewritten to get rid of an abrupt termination, we can write the intended function of its loop body and then match it to the Searching pattern. This particular loop is also interesting in that it takes two input sequences, one iterated with an index and the other with an iterator, and the values are transformed from $samplers$ to $names$.

It wasn't uncommon to find loops that have multiple output variables, especially secondary, flag types of variables. The final values of some of the output variables are calculated differently using different manipulation functions; a common code pattern of the loop body is to use if-then-else statements to calculate results differently or store them in different state variables. For example, there was a loop that essentially copies values from one collection to another but also counts the number of values copied. We were able to handle such loops by matching them to multiple patterns, one for each output variable, as recommended by the pattern catalog and shown in the example in Section 4.2.

As in the example in Section 4.2 we quickly learned that a loop can be matched to different patterns depending on our view on the granularity of data. The same data can be viewed as a scalar, composite, or collection; this is especially true for encapsulated data with a set of well-defined APIs. For example, we found the following loop in our case study, where both $mantissa$ and $exponent$ are int variables.

```
while ((mantissa & 0x0010000000000000L) == 0) {
  exponent−−;
  mantissa ≫= 1;
}
```

Are values collected into $mantissa$ or are they accumulated? It really depends on our intention of the code and our view of $mantissa$'s value. If an int value is viewed as a sequence of bits, it collects constant bits (1's); otherwise, it accumulates values by multiplying by 2. Note that the selected pattern will also determine the form or structure of the derived intended function, e.g., manipulating $mantissa$'s value as bits or an int value. Another deciding factor would be the easiness of applying the matching pattern and deriving an intended function from it.

Our case study showed the value of defining a pattern in terms of the intended functions of the loop body, not its code structure. Many loops have several interdependent state variables, local or global, that are used to perform complex computations and store the results in intermediate states. Although we may need to trace these intermediate state changes to calculate the code or intended function of the loop body, we don't need to worry about them when matching the loops to patterns and applying the matching patterns to derive intended functions. All we care about is the state changes from the initial state to the final state as stated in the intended functions. The real benefit is the modularity that it supports. The derivation of an intended function is still valid—and thus the derived intended function is correct—when the loop body is replaced with another code that is correct with respect to the intended function of the original loop body. In our case study, for example, it was common for loops to have nested loops. We first figured out the intended functions of nested loops by applying our patterns. Then, the derived intended functions of the nested loops were used in calculating those of the loop bodies of the outer loops, enabling applications of our patterns for the outer loops in a modular fashion.

An interesting lesson we learned is the important of abstraction and the notation to express it. One of the most difficult steps of applying our patterns is to formulate and write the intended

function of a loop body. The difficulty is due to not only the complexity of the computation itself but also expressing it in a way suitable for manipulation. In fact, sometimes the complexity of deriving a detailed, rigorously written intended function of a loop depends heavily on expressing the computation of the loop body at an appropriate abstraction level using an appropriate notation. Writing intended functions at the right abstraction levels is difficult and requires skills and experiences; derivations of intended functions themselves can often be done mechanically.

One weakness of our evaluation is missing measurements on the quality of derived intended functions, e.g., whether they are readable, understandable, and usable in formal treatments of programs. However, we would like to note that more than half of sample while loops matched closely with our patterns and didn't seem to require much effort in deriving their intended functions from the matching patterns. Nevertheless, our study shows that our patterns are helpful in finding the intended functions of while loops. As we became more familiar with the use of the patterns, we also became to guess and determine the intention of loops better and more easily, even without applying the patterns explicitly. In a sense, the patterns provided us with a mental framework or tool for examining and analyzing the loops, and the use of patterns improved our insights and analytical skills. We also found a few common patterns of while loops with specific purposes, e.g., traversing trees for various reasons. It would be interesting to study whether they deserve to be documented as sorts of domain and language-specific specializations of our patterns. They capture knowledge in a specific domain, but their usefulness will be determined in part by their generality and variability in order to be instantiated for various loops in the domain.

6. RELATED WORK

No published research work was found on deriving intended functions of loop control structures systematically. It's perhaps partly because Cleanroom-style functional program verification is not well-known. The only closely related work is Stavely's hints on how to write intended functions for while loops in isolation, without their initialization [35, Section 4.4]. His hints include such suggestions as studying the sequence of values stored in program variables as a loop iterates, generalizing the intended functions of an initialized loop, and adapting the intended function of a similar loop. In a way our patterns are a generalization and codification of the last suggestion, adapting the intended function of a similar loop. A loop specification pattern is an abstraction of a collection of similar loops that can be reused by being adapted or instantiated to a specific situation or loop.

Below we mention few recent, noticeable work in three areas of broadly related research: loop invariants, property specification patterns, and source code analysis. The amount of research work done on a similar problem in Hoare logic—finding loop invariants—is huge, spreading over several decades. There exists a rich set of techniques and tools, including both static and dynamic approaches based on execution traces, preconditions, postconditions, theorem proving, etc (see [18, Section 5]) and [34, Section 7]). In an axiomatic approach, loop invariants play a cardinal role in the proofs of loop control structures, for full verification generally requires equipping each loop with a loop invariant. They are also the biggest challenge to full automation of formal analysis and verification of programs because they cannot be computed through simple rules. Finding a sound and useful loop invariant usually requires a programmer's invention relying on skills and experiences. Furia et al classified loop invariants over a range of fundamental and important algorithms, including searching, sorting, and arithmetics [18]. They identified two different dimensions for their classification: the role of the invariant with respect to the postcondition (essential and bounding) and the transformation technique that yields the invariant from the postcondition (constant relaxation, uncoupling, term dropping, aging, and backward substitution) [19]. Their classification can be very useful in understanding the loop invariant of an algorithm, however unlike our work it doesn't provide a reusable pattern that can be instantiated to derive an invariant for a loop. An alternative to requiring a programmer to formulate a loop invariant is to automatically infer one from code. Aponte et al presented an approach for automatically generating loop invariants over non-nested loops manipulating arrays [2]. In their approach, the loop body is first translated into conditional

concurrent assignments (similar to Dijkstra's guarded commands), which are then matched to code patterns through static analysis. Each code pattern is associated with a *local invariant*, an invariant that refers only to variables modified locally. Local invariants are composed to produce an inductive invariant of the complete loop. They defined five categories of code patterns corresponding to simple but frequently used loops over scalar and array variables, such as search, scalar update, scalar integration, array mapping, and array exchange. Their patterns are very specific and specified in terms of code structures so that corresponding local invariants can be defined. The role of a local invariant is similar to that of the intended function of the loop body in our approach; both are strengthened or promoted to cover the complete loop. Furia and Meyer suggested to use not just the code of a loop but its postcondition as the basis for invariant inference, for an invariant of a loop is a weakened form of its postcondition [19]. Their algorithm mutates a postcondition using various heuristics to find a loop invariant. Leino and Logozzo described a technique for automatically generating an essential ingredient of proof, loop invariants, and refine them on demand [26]. The idea is that when an automatic theorem prover fails a proof of a verification condition, an abstract interpreter is invoked on the loops along with the program traces to find stronger loop invariants that will allow the theorem prover to make more progress toward a proof. It allows a gradual increase in the level of precision used by the abstract interpreter and thus generation of loop invariants that are specific to a subset of a program's executions. Recent work has shown that it is possible to infer assertions such as class invariants automatically from program executions. Ernst et al developed a system called Daikon that can dynamically detect a likely program invariant, a property that holds at a certain point or points in a program [11, 12, 31]. The system runs a program, observes the values that the program computes to find properties that were true over the observed executions. Interestingly, however, Polikarpova et al showed that tools like Daikon can be used to strengthen programmer-written assertions, but cannot infer all assertions that programmers write [32].

Since the software design pattern becomes popular and widely used, similar ideas begin to be applied to formal requirement specifications of software systems. In particular, motivated by the inability, for non-experts, to express their requirements using the property specification languages supported by formal verification tools, many researchers have proposed or developed specification pattern systems to facilitate the construction of formal specifications [1, 10, 21, 24]. However, unlike our patterns for source code level specifications, these patterns are mostly described in some forms of temporal logic for specifying various types of system level properties by translating or writing formal specifications from informal or natural language descriptions. The pioneering work on applying the idea of software design patterns to formal specifications is that of Dwyer et al [10]. They developed a set of property specification patterns for finite-state verification like model checking. A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state or event sequences in a finite-state model of a system. It describes the essential structure of some aspect of a system's behavior and provides expressions of this behavior in a range of common formalisms, including quantified regular expressions and various temporal logics such as linear temporal logic (LTL) and computation tree logic (CTL). Mondragon et al introduced composition propositions to allow multiple events or conditions in specification patterns [29]. Konrad and Cheng defined real-time specification patterns as well as a structured English grammar to facilitate the understanding of the meaning of a specification [24]. They developed a stepwise process and a tool suite for deriving and instantiating system properties in terms of their natural language representations [23]. Bid et al. also proposed specification patterns to express real-time requirements for reactive systems [1]. There are also specification patterns formulated in a probabilistic temporal logic for probabilistic verification techniques to ensure software quality requirements [21].

The work on source code analysis is interesting, for some may be adapted to improve our approach, e.g., to partially automate the derivation of the intended functions. The automated and semi-automated analysis of source code has been a topic of research for more than several decades [6]. A static loop analysis is a source code analysis technique for automatically extracting, finding or deriving a wide range of useful information about loop, such as loop iteration counts, code execution frequencies, infeasible paths, and loop bounds. The derived information can be used for various

purposes such as loop optimizations and worst-case execution time estimation. Several techniques have been proposed for fully automating static analysis of loops at source code level, including pattern-based approach, source code annotation, data flow analysis, abstract interpretation, program slicing, and invariant analysis (e.g., [25, 27]). We believe that some of the derived information from loop analysis be useful in our approach, e.g., data flow analysis can provide data dependency information that can be utilized to define the basic structures of the expressions appearing in intended functions.

7. CONCLUSION

We presented specification patterns to address the problem of formulating candidate or likely specifications of loop control structures for formal analysis and verification of programs. Any non-trivial program contains loop control structures such as while, for, and do statements, and formal verification of the program requires to equip each loop with a candidate specification. In functional program verification, a candidate specification for a loop is an intended function that expresses the final values of variables as a function of initial values; an intended function documents the net effect of a section of code on data from entry and exit. A candidate intended function for a loop plays a crucial role in formal verification of the loop because it becomes an induction hypothesis in an inductive proof of the loop. However, formulating a likely intended function of a loop is one of the biggest challenges in a correctness proof of the loop, mostly relying on one's skills and experiences, for there is no simple rule to compute it.

Fortunately, many intended functions of loops exhibit certain common flavors or characteristics. Knowing these flavors or characteristics could therefore provide help in formulating likely intended functions of loops. Inspired by the work on software design patterns, we identified these common flavors of intended functions and documented them as reusable specification patterns, from which intended functions of loops can be derived systematically. Loops are most commonly used to iterate over a certain sequence of values and manipulate it, typically one value at a time. One distinguishing feature of our patterns is to promote the intended function of the loop body, manipulating individual values, to the whole sequence iterated over by a loop to define the intended function of the whole loop. Our patterns include Accumulating, Searching, Selecting and Collecting along with numerous variations.

Our specification patterns are compositional and hierarchical. A pattern can be decomposed along the four, orthogonal dimensions of loop analysis: value acquisition, value manipulation, loop termination, and result storage. As a consequence, a new pattern can be assembled by selecting an appropriate combination of the values from these dimensions. Our patterns can also be classified into a pattern hierarchy. A generalized pattern is applicable to a wide range of loops, but its specification is more abstract and thus provides less help in deriving a detailed intended function from its application. A specialized pattern, on the other hand, is more specific with limited applicability but provides more help in deriving a detailed intended function. The pattern hierarchy is extensible in that one can easily introduce a new pattern by refining or specializing the value manipulation function of an existing pattern. The pattern hierarchy allows one to match patterns, starting from more general patterns and moving down to more specific ones. A case study indicates that our patterns are applicable to a wide range of programs from systems programming to scientific and business applications.

There are several contributions of our work. The four, orthogonal loop analysis dimensions provide an excellent conceptual framework for examining loops systematically. They can be used not only for general understanding of loops but also for composing new patterns and finding matching patterns for loops. Our pattern catalog provides a set of reusable loop specifications that can be matched to and instantiated to derive intended functions of loops systematically. Unlike previous work on specification patterns, our patterns are for deriving source code-level specifications for formal analysis and verification of programs; they provide a solution to the problem of formulating candidate or likely specifications of loop control structures for various formal treatments of code containing loops. Another uniqueness of our work is the idea of promoting

the manipulation of individual elements to the whole sequence to define a pattern, resulting in a uniform pattern structure and facilitating an easy introduction of new patterns.

ACKNOWLEDGEMENT

This work was supported in part by NSF grant DUE-0837567. Any opinion, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

1. Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. Real-time specification patterns and tools. In *Formal Methods for Industrial Critical Systems*, volume 7437 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-32469-7_1.
2. Virginia Aponte, Pierre Courtieu, Yannick Moy, and Marc Sango. Maximal and compositional pattern-based loop invariants. In *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-32759-9_7.
3. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
4. Aditi Barua and Yoonsik Cheon. A catalog of while loop specification patterns. Technical Report 14-65, Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, September 2014.
5. Aditi Barua and Yoonsik Cheon. Finding specifications of while statements using patterns. In *New Trends in Networking, Computing, E-learning, Systems Sciences, and Engineering*, volume 312 of *Lecture Notes in Electrical Engineering*, pages 581–588. Springer International Publishing, 2015. DOI: DOI 10.1007/978-3-319-06764-3_75.
6. David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society. DOI: 10.1109/FOSE.2007.27.
7. Yoonsik Cheon. Functional specification and verification of object-oriented programs. Technical Report 10-23, Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, August 2010.
8. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software: Practice and Experience,*, 35(6):583–599, May 2005. DOI: 10.1002/spe.649.
9. Yoonsik Cheon, Cesar Yeep, and Melisa Vela. The CleanJava language for functional program verification. *International Journal of Software Engineering*, 5(1):47–68, January 2012.
10. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM. DOI: 10.1145/298595.298598.
11. M.D. Ernst, J. Cockrell, William G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. DOI: 10.1145/302405.302467.
12. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. DOI: 10.1016/j.scico.2007.01.015.
13. Apache Software Foundation. Apache HTTP Server Project. https://httpd.apache.org. Accessed: 2015-12-18.
14. Apache Software Foundation. Apache Jmeter. https://jmeter.apache.org. Accessed: 2015-12-18.
15. Apache Software Foundation. Apache Syncope. https://syncope.apache.org. Accessed: 2015-12-18.
16. Apache Software Foundation. Chukwa. https://chukwa.apache.org. Accessed: 2015-12-18.
17. Apache Software Foundation. Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math. Accessed: 2015-12-18.
18. Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys*, 46(3):34:1–34:51, January 2014. DOI: 10.1145/2506375.
19. Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, pages 277–300. Springer-Verlag, Berlin, Heidelberg, 2010. DOI: 10.1007/978-3-642-15025-8_15.
20. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
21. Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. ACM. DOI: 10.1145/1368088.1368094.
22. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.
23. Sascha Konrad and Betty H. C. Cheng. Facilitating the construction of specification pattern-based properties. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, pages 329–338. IEEE Computer Society, 2005. DOI: 10.1109/RE.2005.29.
24. Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM. DOI: 10.1145/1062455.1062526.

25. Eric Larson. Program analysis too loopy? set the loops aside. *IET Software*, 7(3):131–149, June 2013. DOI: 10.1049/iet-sen.2012.0048.
26. K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 119–134, Berlin, Heidelberg, 2005. Springer-Verlag. DOI: 10.1007/11575467_9.
27. Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society. DOI: 10.1109/CGO.2009.17.
28. Harlan D. Mills, Michael Dyer, and Richard Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987. DOI: doi:10.1109/MS.1987.231413.
29. Oscar Mondragon, Ann Q. Gates, and Steven Roach. Prospec: Support for elicitation and formal specification of software properties. In *RV 2003, Run-time Verification (Satellite Workshop of CAV '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 67–88. Elsevier, 2003. DOI: doi:10.1016/S1571-0661(04)81043-0.
30. Robert Oshana. Tailoring Cleanroom for industrial use. *IEEE Software*, 15(6):46–55, November 1998. DOI: 10.1109/52.730840.
31. Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 23–32, New York, NY, USA, 2004. ACM. 10.1145/1041685.1029901.
32. Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 93–104, New York, NY, USA, 2009. ACM. DOI: 10.1145/1572272.1572284.
33. Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering*. Addison Wesley, February 1999.
34. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pages 574–592, Berlin, Heidelberg, 2013. Springer-Verlag. DOI: 10.1007/978-3-642-37036-6_31.
35. Allan M. Stavely. *Toward Zero Defect Programming*. Addison-Wesley, 1999.
36. Marvin V. Zelkowitz. A functional correctness model of program verification. *Computer*, 23(11):30–39, November 1990. DOI: 10.1109/2.60878.
37. Xiaoyan Zhu, E. James Whitehead, Caitlin Sadowski, and Qinbao Song. An analysis of programming language statement frequency in C, C++, and Java source code. *Software: Practice and Experience*, 45(11):1479–1495, 2015. DOI: 10.1002/spe.2298.