Exploring the Generality of a Java-based Loop Action Model for the Quorum Programming Language

PRECIM-RESEARCH REPORT

PREETHA CHATTERJEE

Department Computer & Information Sciences, University of Delaware

Email: preethac@udel.edu



Contents

1	Intro	oductio	on Control of the Con	3
2	2.1 2.2	2.1.12.1.22.1.32.1.4	pping A Model of Loop Actions Terminology Features Action Identification Model Limitations Im Language Introduction Language Basics	5 6 8 9 9
3	Buil	ding a	Loop Action Model for Quorum	10
	3.1	Overv	iew	10
	3.2	Manua	al identification of loop-if and mapping to Action Identification Model	11
		3.2.1	Mapping Java loop structure to Quorum loop structure	11
		3.2.2	Mapping Java if condition to Quorum if-condition structure	13
		3.2.3	Mapping Java loop-if structure to Quorum loop-if structure	13
		3.2.4	Mapping Java Feature Vectors to Quorum Feature Vectors	14
		3.2.5	From Loop to Feature Vector: An example	15
		3.2.6	Evaluation	15
	3.3	Develo	oping a tool for Automatic Action Identification	16
		3.3.1	Quorum Language Grammar	
		3.3.2	ANTLR	17
		3.3.3	Quorum Abstract Syntax Tree (AST)	18
		3.3.4	AST-walker-based Feature Extractor	19
4	Ana	lysis o	f the Generality of The Loop Action Model	22
	4.1	Applyi	ng the Loop Action Model for Quorum	22
	4.2	Evalua	ation of Automatically Identified Actions in Quorum	23
		4.2.1	Methodology	23
		4.2.2	Results	23
		4.2.3	Comparing the results: Java vs. Quorum	25
	4.3	Threa	ts to Validity	25
5	Rela	ated Wo	ork	25
6	Con	clusio	n and Future Work	26

Exploring the Generality of a Java-based Loop Action Model for the Quorum Programming Language

Abstract

Many algorithmic steps require more than one statement to implement, but not big enough to be a method (e.g., add element, find the maximum, determine a value, etc.). These steps are generally implemented by loops. Internal comments for the loops often describe these intermediary steps, however, unfortunately a very small percentage of code is well documented to help new users/coders. As a result, information at levels of abstraction between the individual statement and the whole method is not leveraged by current source code analyses, as that information is not easily available beyond any internal comments describing the code blocks.

Hence, this project explores the generality of an approach to automatically determine the high level actions of loop constructs. The approach is to mine loop characteristics of a given loop structure over the repository of the Quorum language source code, map it to an (already developed for Java) action identification model [1], and thus identify the action performed by the specified loop. The results are promising enough to conclude that this approach could be applied to other programming languages too.

1 Introduction

In a software program, typically, multiple algorithmic steps combine to form a method. Examples are: Finding an element that satisfies some condition, comparing all pairs of corresponding elements from two collections. Although the algorithmic steps (which are the building blocks of the method) are small steps, they generally take more than one line of code to implement. The internal comments usually describe the actions of these algorithmic steps. However, descriptive internal comment, which would help the reader understand the code in a easier and better way, are very rare [10], [11]. Adding to the lack of comments is the inadequacies of relying on names in the code. Not always can the method names and the variable names be defined in such a manner that the complete logic of the steps performed is subjectively clear from just "reading the names".

Thus, information for the whole method, with details of individual algorithmic steps is not captured by present source code analyses. The main reason is the lack of internal comments. The current software tools performing source code analyses treat this problem in one of three ways. The first approach is to treat each method as a single unit, thus the source code analyses takes into account only the overall task performed by the method and not the detailed algorithmic steps which create it. The second approach is to treat the method as a set of individual statements. For the second approach, some documentation generators for method summaries process the methods as a set of individual statements and then select a subset of statements for which to generate a method summary [12]. The third approach is to use methods as a "bag of words" and select a subset of words for the summary [17]. While Sridhara et.al [14] used a set of templates to identify

high level abstractions and generate summary comments for methods, Xiaoran et al. [1] developed an action identification model to do the same without using manually created templates, and thus implemented a more flexible approach.

This project focuses on exploring the generality of the Java-based Loop Action Model developed by Wang et.al [1] for the Quorum Programming Language [4]. Quorum is a programming language designed specially for visually impaired middle and high school students. The objective is to investigate summarizing loops in Quorum, to identify the higher level abstraction of the action being performed by the loops.

The steps of this project are:

- 1. Manually identify loop-if structures (loops that contain exactly one conditional statement "if-statement", which is also the last lexical statement within the loop body) from sample Quorum code, for at least 25 loops, and extract the corresponding feature vectors.
- 2. Manually map the identified loop-ifs with the already developed action identification model [1] and validate the results.
- 3. Develop an automatic feature extraction system by using ANTLR(parser generator) [2].
- 4. Evaluate the output of the automatic tool/system.

As Quorum is a comparatively new programming language, the repository of sample projects available for the purpose of research is small. For this project, the Quorum language compiler and its standard library files are used as the subjects of study. So understandably, the results could not be evaluated on different coding practices from different developers. To address that concern, after successful implementation of the project, an unbiased question-answer survey was conducted, to evaluate the correctness of the results.

Contributions: The main contributions of the project include:

- an automatic tool to identify the high level actions implemented by Quorum loops.
- demonstration of the feasibility of using the Java Loop Action Model to summarize loops in Quorum.
- evaluation results from human judgement study that indicate strong positive opinion of the tool's effectiveness in automatically identifying high level actions for these loop structures.

Beyond showing generality of the loop action model, the perceived impact of the work is also to help blind programmers by providing them with the summary rather than reading the detailed loop code. This project has the potential to increase the effectiveness of code search tools and comment generator tools by providing the action phrase with the associated loop. It would also help to obtain a better comprehension of code, especially for blind readers.

2 Background

2.1 Developing A Model of Loop Actions

Motivation for this project comes from existing work by Xiaoran Wang, Dr. Lori Pollock and Dr. Vijay Shanker on "Developing a Model of Loop Actions by Mining Loop Characteristics from a Large Code Corpus" [1] - which involved identifying the higher level abstraction of the action being performed by a particular loop structure in Java based on their structure, data flow and linguistic characteristics. Their approach (Figure 1) was to first identify action units (a code block that consists of a sequence of consecutive statements that logically implement a high level action) that are implemented by loop structures, characterize the loops as feature-value pairs to generate the loop feature vector (set or sequence of feature values) and then develop a model (action identification model) that can associate actions with loops based on their loop feature vectors.

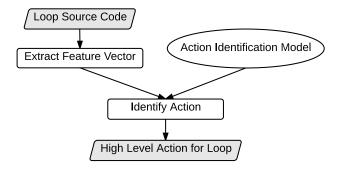


Figure 1: Action Identification Process

2.1.1 Terminology

The following terminology is used to describe the features, that are used to determine the loop actions. This terminology was developed by Wang et al. [1].

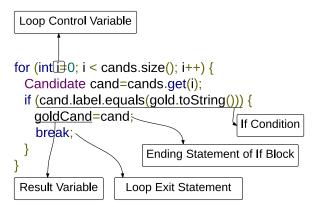


Figure 2: Example of terminology

Figure 2 shows an example loop in Java to demonstrate the terminology used in developing the feature vector.

- **If Condition:** The if condition refers to the conditional expression in the if statement of the loop.
- Loop Exit Statement: Loop exit statements transfer control to another point in the code by exiting when control reaches the loop exit statement, such as a break or return.
- Ending Statement of if block: As the last statement inside a loop-if is an if, the last executed statement of the loop is the last statement inside the if-block, which is referred as the ending statement. If the last executed statement of the loop is a branching statement like return, break or throw, then the statement immediately preceding the branching statement is considered to be the ending statement.
- Loop Control Variable: The loop control variable is the variable defined in the loop condition.
- Result Variable: Result variable captures the resulting value of the loop's action (if one exists). If the ending statement is an assignment, the result variable is the left-hand-side variable. If the ending statement is an object method invocation, it is the object that invokes the method.

2.1.2 Features

The features of loop-ifs separated into features related to ending statements and features related to the if condition are described [1].

Features related to Ending Statement:

- **F1: Type of ending statement -** If the last executed statement of the loop is a branching statement like return, break or throw, then the statement immediately preceding the branching statement is considered to be the ending statement.
- **F2: Method name of ending statement method call -** If the ending statement is a method invocation, the first verb comprising the method name is extracted. In table 1, they have however used method names occurring in the 100 most frequent loop-ifs.
- **F3: Elements in collection get updated -** F3=1 is set, when the result variable is the loop control variable; otherwise, F3=0.
- **F4:** Usage of loop control variable in ending statement The loop condition determines the maximum number of iterations that will be executed. F4=0 is set, when the loop control variable never appears in the ending statement; F4=1, when the loop control variable is directly used in the ending statement; F4=2, when the loop control variable is on a def-use chain to a use in the ending statement.
- **F5: Type of loop exit statement -** F5 denotes if there is a control flow disruption in the action unit, and if there is, then the type (break, return, return boolean, return object or throw).

Features related to the if-condition:

- **F6: Multiple collections in if condition -** This feature is a boolean that indicates whether multiple collections are compared in the if condition. F6=1 if there are two synchronized collections in the if condition; otherwise, F6=0.
- **F7: Result variable used in if condition -** F7=1 is set, if the result variable appears in the if condition; otherwise, F7=0.
- **F8: Type of if condition -** If the if condition is a numeric comparison("<" , ">", "<=" or ">=") then F8=1 is set. Otherwise, if the type of if condition is a boolean value returned from a user defined method, then F8=2.

	Table 1	details the	potential	values	for each	loop-if feature.
--	---------	-------------	-----------	--------	----------	------------------

Label	Feature	Possible Values and their Semantics	
F1	Type of ending statement	0: none 1: assignment 2: increment 3: decrement 4: method invocation	
		5: object method invocation 6: boolean assignment	
F2	Method name of ending statement	0: none 1: add 2: addX 3: put 4: setX 5: remove	
	method call		
F3	Elements in collection get updated	0: false 1: true	
F4	Usage of loop control variable in ending	0: not used 1: directly used 2: used indirectly through data flow	
	statement		
F5	Type of loop exit statement	0: none 1: break 2: return 3: return boolean 4: return object 5: throw	
F6	Multiple collections in if condition	0: false 1: true	
F7	Result variable used in if condition	0: false 1: true	
F8	Type of if condition	1: >/=/<= 2: others	

Table 1: Semantics of Feature Values

From Loop to Feature Vector: An example

The feature vector for the example code fragment in Figure 2 is:(F1:1, F2:0, F3:0, F4:2, F5:1, F6:0, F7:0, F8:2). F1 indicates that the ending statement is an assignment. F2 indicates there is no method name from an ending method call. F3 indicates that not every element in the collection is updated. F4 indicates that the loop control variable is on the def-use chain to a use in the ending statement. F5 indicates that the type of loop exit statement is a break. F6 indicates that there is only one collection in if condition. F7 indicates that the result variable is not used in the if condition. F8 indicates that the type of the if condition is not numeric comparison.

2.1.3 Action Identification Model

Action Feature	F1	F2	F3	F4	F5	F6	F7	F8
count	2				0			
determine	0				2,3			
determine	6				0,1			
max/min	1			1,2	0	0	1	1
find	1			1,2	1,2,4			
find	0				4			
сору	5	1	0	1	0			
ensure					5			
compare					3	1		
remove	5	5	1	1	0			
get	5	1,3	0	2	0			
add	5	2	0	1,2	0			
set_one	5	4	0	1,2				
set_all	5	4	1	1,2	0			

Table 2: Action Identification Model

Label	Action Phrase		
count	count the number of elements in a collection that satisfy some condition		
determine	determine if an element of a collection satisfies some condition		
max/min	find the maximum/minimum element in a collection		
find	find an element that satisfies some condition (other than max/min)		
copy copy elements that satisfy some condition from one collection to anoth			
ensure	ensure that all elements in the collection satisfy some condition		
compare	compare all pairs of corresponding elements from two collections		
remove	remove elements when some condition is satisfied		
get	get all elements that satisfy some condition		
add	add a property to an object		
set_one	set properties of an object using objects in a collection that satisfy some condition		
set_all	set a property for all objects in a collection that satisfy some condition		

Figure 3: Actions with their verb phrase descriptions

To characterize the high level action performed by a specific feature vector, Wang et al. [1] examined several loops corresponding to that loop feature vector that had comments associated with them, extracted the verbs from comments, computed verb distributions for vectors, clustered vectors based on verb distribution and selected representative action for each cluster. Thus, they developed an action identification model, where each row shows an action and its corresponding set of feature values, as shown in Table 2. For example, if a loop has value 0 for F1 and 2 or

3 for F5, or value 6 for F1 and 0 or 1 for F5, then the model will label this loop with the action "determine". Figure 3 shows the action phrases for each of the actions.

2.1.4 Limitations

The authors note several limitations of the work so far:

- The only loop formats considered in Java were: for, enhanced-for, while or do-while.
- The paper focused on loops that contain exactly one conditional statement (if-statement), which is also the last lexical statement within the loop body (a loop-if structure).
- Nested loops were not in the scope.
- The action identification table was developed based on the 100 most frequent loop feature vectors in the data set used for the project, and not for all possible loop feature vectors.

2.2 Quorum Language

2.2.1 Introduction

Quorum is a programming language that is built, keeping in the mind the problems faced by the blind students to learn and use computer programming languages in general. To quote Dr. Andreas Stefik, one of the inventors of Quorum, "The blind and visually impaired community is significantly underrepresented in computer science. Students who wish to enter the discipline must overcome significant technological and educational barriers to succeed. While much work has been dedicated to helping the blind use various computer technologies, more research is needed on finding ways to make it easier for blind users to obtain high-paying and meaningful careers. Indeed, with 61% of working adults (aged 16 to 64) with vision loss out of the work force, and with households that include a blind member having a significantly higher rate of poverty, creating more opportunities for this group of individuals is sorely needed. "[7].

Quorum started as an interpreted language originally designed to be easier to hear through screen readers for blind or visually impaired users. Eventually, it became a general purpose programming language designed for any user. Current versions compile to Java Bytecode and run on the Java Virtual Machine, similar to JRuby, Jython, or Scala. Quorum 3.0 also compiles to JavaScript and can be run from the web [4].

2.2.2 Language Basics

Quorum is an object-oriented programming language which has a general purpose type system, with generics for containers (e.g., arrays, hash tables, lists). Quorum also has a standard library, which contains many additions to the language, including math libraries, web components, and a game engine [4].

As this project explores whether the Java loop action model can be used for Quorum, it is important to understand the loop structures in Quorum, which are different from that of Java. First there is no for-loop. Instead there are three different loop types as follows:

• repeat <expression> times:

```
integer a = complicatedMathAction()
integer b = anotherComplexAction()
repeat (b / a - (b + 5)) times
end
```

repeat while <expression>

```
integer a = 0
repeat while a < 15
   a = a + 1
end</pre>
```

repeat until <expression>

```
integer a = 0
repeat until a < 15
   a = a + 1
end</pre>
```

In this case, since a is less than 15 this loop will execute 0 times.

Source Code for the Quorum project can be found at the Quorum Bitbucket page [3] at:

```
https://bitbucket.org/stefika/quorum-language.
```

3 Building a Loop Action Model for Quorum

3.1 Overview

The goal of this project is to explore the generality of the Loop Action Model & Feature Vector approach to identify high level actions for loop-ifs in Quorum. For Java, the action identification model is in table 2. The overall process of this task is shown in Figure 4. To investigate generality to Quorum, we need to investigate whether and how the same loop features can be extracted from the Quorum loop-ifs and whether the same identification model is applicable in Quorum codes.

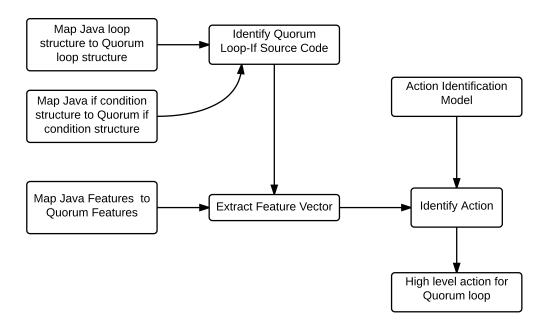


Figure 4: From Java to Quorum

Thus, we need to map loop formats in Java (for, enhanced-for, while or do-while) to loop formats in Quorum (repeat times, repeat while and repeat until), if conditions in Java to if-conditions in Quorum and each feature value in Java to the equivalent in Quorum. After the mappings, Quorum Loop-if source code can be identified. The action identification model is then referenced to determine the high level action associated with the loop's feature vector.

3.2 Manual identification of loop-if and mapping to Action Identification Model

3.2.1 Mapping Java loop structure to Quorum loop structure

Quorum has no "for" or "enhance-for" loops. Instead the loops are of types: "repeat <expression> times", "repeat while <expression>" and "repeat until <expression>", all of which are considered in this project.

repeat <expression> times:

There is no repeat <expression> times loop in Java. The code snippets below show the same logic written in Java (using a for-loop) and Quorum. Differences are bolded in each of them. The variable declaration and print statements are almost similar in both of the languages, but the loop structures are entirely different in this case.

Figure 5: Comparison of repeat <expression> times loop structures in Java (Left), and Quorum (Right)

• **repeat while <expression>:** Repeat while <expression> loops in Quorum are quite similar to while loops in Java. The code snippets below show the same logic written in Java (using a while-loop) and Quorum. The only major difference is the loop-syntax ("()", "", keywords "repeat" and "end"), bolded below.

Figure 6: Comparison of repeat while <expression> loop structures in Java (Left), and Quorum (Right)

 repeat until <expression>: There is no repeat until <expression> loop in Java. The code snippets below show the same logic written in Java (using a for-loop) and Quorum. Differences are bolded in each of them. As before, the loop structures are entirely different in this case as well.

```
Java: Quorum: int a = 17; for (a = 17; a >= 15;) integer a = 17 repeat until a < 15 a = a-1; System.out.println (a); output a end
```

Figure 7: Comparison of repeat until <expression> loop structures in Java (Left), and Quorum (Right)

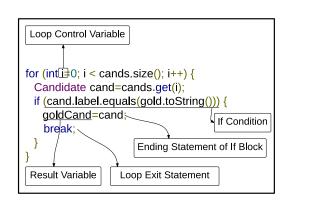
3.2.2 Mapping Java if condition to Quorum if-condition structure

The "if conditional" in Quorum is almost similar syntactically with "if condition" in Java. So mapping Java if-conditions to Quorum if-conditionals is straight forward. The code snippets below show the same logic written in Java and Quorum. The only differences are the use of "()", ";" and "end".

Java:	Quorum:		
if (a > 100) c = 1; elseif (b == 100) c = 2; else	if a > 100 c = 1 elseif b = 100 c = 2 else c = 3		
c = 3;	end		

Figure 8: Comparison of if structures in Java (Left), and Quorum (Right)

3.2.3 Mapping Java loop-if structure to Quorum loop-if structure



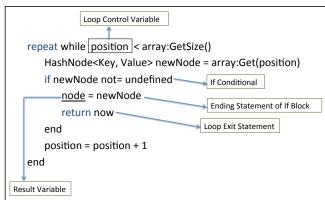


Figure 9: Comparison of Loop-if structures in Java (Left), and Quorum (Right)

Figure 9 compares sample loop-if structures in Java and Quorum. The types of loop structures used in the examples in figure 9, are different - Java (for-loop) and Quorum (repeat while <expression>). Hence, the Loop Control Variable ('i') in the for-loop in Java is mapped to the Loop Control Variable ('position') in the <expression> for Quorum. "If-conditions", "Ending Statement of if-block", "Loop Exit Statement" and "Result Variable" are similar in both Java and Quorum .The only syntactic differences are in the use of "()", "{}", ";", ":", and keyword "end".

However, within the Quorum repeat loop, there is an extra lexical statement right after the if-conditional: "position = position + 1". As per the definition, a loop-if structures in Java is a loop that contain exactly one conditional statement (if-statement), which is also the last lexical statement within the loop body. But if a closer look is taken into the Quorum code snippet, it is clear that

"position" being the loop-control variable, the statement after the if-conditional is doing nothing else, but actually increasing the counter of the repeat loop, same as what "i++" is doing in the Java for-loop. So in this example, though the Quorum and Java code snippets are syntactically different, but are semantically both "loop-if" structures.

3.2.4 Mapping Java Feature Vectors to Quorum Feature Vectors

The features of Java loop-ifs, described in section 2.1.2, are now mapped to that of Quorum.

- F1: Type of ending statement Similar to Java, the different types of ending statements (if present) are: assignment, increment, decrement, method invocation, object method invocation and boolean assignment. Thus this feature is identified the same in Java and Quorum. The syntactic type of ending statement can be a strong indicator of the overall purpose of the loop. If there is no ending statement in the loop, F1 is set to 0: none.
- **F2: Method name of ending statement method call** Similar to Java, if the ending statement is a method invocation, the first verb comprising the method name is extracted. Thus this feature is identified the same in Java and Quorum. The values for this feature are none, add, addX, put, set and remove.
- **F3:** Elements in collection get updated Similar to Java, F3=1 is set, when the result variable is the loop control variable; otherwise, F3=0. Thus this feature is identified the same in Java and Quorum.
- F4: Usage of loop control variable in ending statement Similar to Java, F4=0 is set, when the loop control variable never appears in the ending statement; F4=1, when the loop control variable is directly used in the ending statement; F4=2, when the loop control variable is on a def-use chain to a use in the ending statement. Thus this feature is identified the same in Java and Quorum.
- F5: Type of loop exit statement F5 denotes if there is a control flow disruption in the action unit, and if there is, then the type. For Quorum, we have the following types return, return boolean, return object. Type4 or "break" and Type5 or "throw" in Java do not exist in Quorum, hence they are not possible values here.
- F6: Multiple collections in if condition Collections in Java is equivalent to Containers in Quorum. This feature is a boolean that indicates whether multiple collections (containers) are compared in the if condition. F6=1 if there are two synchronized containers in the if condition; otherwise, F6=0.
- F7: Result variable used in if condition Similar to Java, F7=1 is set, if the result variable appears in the if condition; otherwise, F7=0. Thus this feature is identified the same in Java and Quorum.
- **F8: Type of if condition -** Similar to Java, if the if condition is a numeric comparison("<", ">", "<=" or ">=") then F8=1 is set. Otherwise, for eg. if the type of if condition is a boolean

value returned from a user defined method, then F8=2. Thus this feature is identified the same in Java and Quorum.

To summarize, with the slight modifications as mentioned above, only features F5 and F6 change for Quorum. The modified table from Wang et al. [1] for "Semantics of Feature Values" for Quorum is as follows:

Label	Feature	Possible Values and their Semantics	
F1	Type of ending statement	0: none 1: assignment 2:increment 3:decrement 4:method invocation 5:object method invocation 6: boolean assignment	
F2 Method name of ending statement method call		0:none 1:add 2:addX 3:put 4:setX 5:remove	
F3	Elements in collection get updated	0: false 1: true	
F4	Usage of loop control variable in ending statement	0: not used 1:directly used 2:used indirectly through data flow	
F5	Type of loop exit statement	0:none 2:return 3:return boolean 4:return object	
F6	Multiple collections(containers) in if condition	0: false 1: true	
F7	Result variable used in if condition	0: false 1: true	
F8	Type of if condition	1: >/=/<= 2: others	

Table 3: Semantics of Feature Values for Quorum (Differences from Java are bolded)

3.2.5 From Loop to Feature Vector: An example

A feature vector for a given loop-if is constructed by extracting the features F1 through F8 from the loop's source code representation using simple static analysis. The feature vector for the example code fragment of Quorum in Figure 2 is: (F1:1, F2:0, F3:0, F4:2, F5:4, F6:0, F7:0, F8:2).

F1 indicates that the ending statement is an assignment. F2 indicates there is no method name for an ending statement method call. F3 indicates that no element in a collection is updated. F4 indicates that the loop control variable is on indirect use in the ending statement. F5 indicates that the loop exit statement is returning an object. F6 indicates that there is no collection in if condition. F7 indicates that the result variable is not used in the if condition. F8 indicates that the type of the if condition is not numeric comparison.

Mapping the feature vector to the action identification model in Table 2, this loop action is identified as 'find'. Thus, the action can be identified as "find an element that satisfies some condition".

3.2.6 Evaluation

After the initial phase of manually identifying loop-if structures in Quorum and mapping to the Action Identification Model in Table 2, a study was conducted to evaluate the accuracy of this approach for Quorum. A set of 10 sample Quorum code snippets containing loop-if structures,

was given to Xiaoran Wang, the first author of the paper - "Developing a Model of Loop Actions by Mining Loop Characteristics from a Large Code Corpus" [1]. Loops were randomly selected - 60% of sample snippets were loop-ifs for which actions could be identified by the approach. The evaluator was asked to determine the corresponding feature vectors and the identified actions. His expert results were compared with the results from manually applying the approach, and there was a 100% match of the results.

3.3 Developing a tool for Automatic Action Identification

The objective of developing a tool for Automatic Action Identification is to determine if given a piece of Quorum source code (containing a loop-if) as input, the tool is able to extract the corresponding feature vector values and identify the high level action performed by the loop. The overall approach to automatic action identification is shown in figure 10.

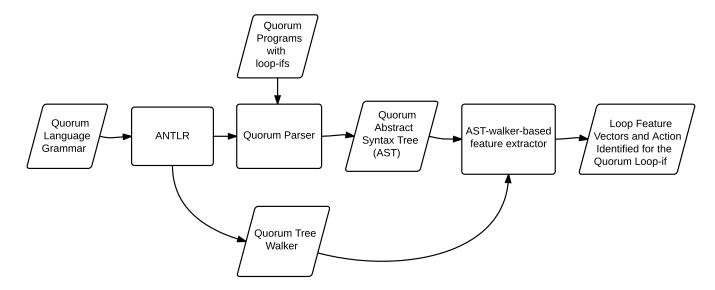


Figure 10: Automatic Action Identification of a Quorum loop-if

3.3.1 Quorum Language Grammar

A grammar formally defines the syntax rules of a language. The first step in the process for automation was to understand the basic architecture of the Quorum compiler and identify/extract Quorum language grammar from the source code repository of Quorum at [3]. Figure 11 shows some of the relevant parts(rules) of the Quorum grammar referred to extract the feature vectors of loop-ifs.

```
if_statement
                                                       IF expression
                                                       (elseif_statement)*
                                                       (else_statement
                                                       )?
                                                       END
                                                  elseif_statement
                                                       ELSE_IF
                                                       expression
                                                       block
                                                  else statement
        grammar Quorum;
                                                       ELSE block
        start
                 (package_rule reference+
                                                   loop_statement
                 reference+ package_rule
                 package_rule
                                                       REPEAT (
                 reference+
                                                                (expression TIMES)
                                                           ((WHILE | UNTIL) expression)
            class_declaration EOF
                                                                ) block END
statement:
        solo_method_call
        if_statement
        assignment_statement
        loop_statement
        return_statement
       print_statement
        speak_statement
        check_statement
        alert_statement
solo_method_call
    qualified name (COLON ID)? LEFT_PAREN (expression (COMMA expression)*)? RIGHT_PAREN
       PARENT COLON qualified_name COLON ID LEFT_PAREN (expression (COMMA expression)*)? RIGHT_PAREN
       ME COLON qualified_name (COLON ID)? LEFT_PAREN (expression (COMMA expression)*)? RIGHT_PAREN
```

Figure 11: Relevant Parts of Quorum Grammar

3.3.2 **ANTLR**

ANTLR (ANother Tool for Language Recognition) is a parser generator for reading, processing, executing, or translating structured text or binary files. The latest version of the Quorum compiler uses ANTLR4 backend. From the Quorum language grammar, ANTLR generates a Quorum Parser that automatically builds Quorum Abstract Syntax Trees (AST) [2]. ANTLR also automatically generates Quorum tree walkers that are used to visit the nodes of the ASTs.

3.3.3 Quorum Abstract Syntax Tree (AST)

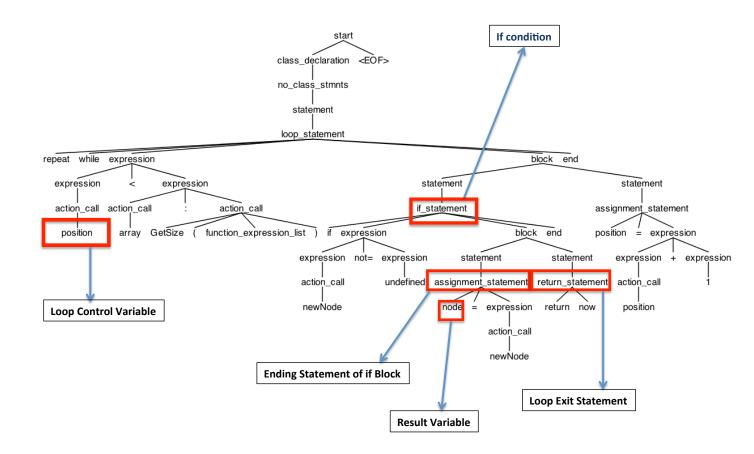


Figure 12: Quorum Parse Tree for an example Quorum loop-if

Figure 12 shows the ANTLR generated parse tree for the Quorum (loop-if) code snippet in Figure 9. The terminologies used for identifying the feature vectors of a loop-if (section 2.1.1) are highlighted in the figure. Each of the terminologies is identified in the context of the ASTs as follows:

- Loop Control Variable: The Quorum code repeat while position < array:GetSize uses a repeat while <expression> loop. So, the loop-control variable "Position" is to be located under the third child node "expression" under parent node "loop_statement". The first and second child nodes of "loop_statement" are "repeat" and "while" respectively.
- If Condition: We are only interested in if conditions inside the loop structure (i.e. a loop-if structure). So, an "if-condition" is to be located under the fourth child node "block" under

parent node "loop_statement". Then, the second last node (as the last node pertains to the increment of loop control variable in this case) is examined for it's type. If it is an "if_statement", the particular Quorum code snippet under consideration is identified to be a "loop-if" structure.

- Loop Exit Statement: The Loop Exit Statement (if exists) should be the last statement inside the "if block". So the parse tree is traversed down the path from the "if_statement" node to its child node "block", down to the last "statement node" under it. This "statement" node is further investigated to identify it's type. If it is a "return_statement", it is concluded that the loop-if structure under consideration contains a "Loop Exit Statement".
- Ending Statement of if Block: The Ending Statement of if block is essentially the last statement inside the "if block". However if a loop exit statement is present, the ending statement would be the second last statement inside the block (same as shown in Figure 12). So the parse tree is traversed down the path from the "if_statement" node to its child node "block", down to the second last "statement" node under it. The type of an Ending Statement could be "assignnment_statement" (as shown above) or it could be a "method_call", depending on the code snippet examined.
- **Result Variable:** The result variable is a part of the "Ending Statement of if block". For most of the cases, the first child node of the "Ending Statement" is the result variable.

3.3.4 AST-walker-based Feature Extractor

My AST-walker-based Feature Extractor tool parses the ANTLR generated parse tree (for the input Quorum code), determines the loop feature vectors, and identifies the action of a Quorum loop-if. Each of the feature vectors for a Quorum loop-if is determined from the AST as described below.

F1: Type of ending statement :

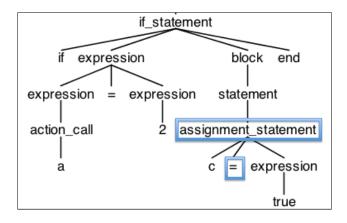


Figure 13: Part of Quorum AST (F1 = 6)

The Feature Extractor tool reads all the statements inside the "if condition" of the loop-if, and identifies the ending statement of the loop, if it exists. If it does, the code reads the parse tree to find out if the statement is of type "assignment"- assignment statements could be processed further to check if it has "increment" or "decrement" operators, or else it could be also a boolean assignment. If the ending statement is not of type "assignment", then the code finds out whether it is of type "solo_method_call" to identify a object method invocation or a simple method call. In figure 13 the block of AST shows an example of F1=6, which means the type of ending statement is boolean assignment.

F2: Method name of ending statement method call :

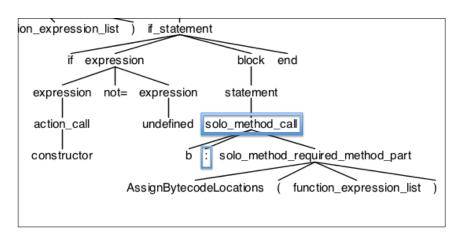


Figure 14: Part of Quorum AST (F2 = " ")

If the ending statement is of type "solo_method_call", the statement is split on the operator ":" to determine the name of the method. In figure 14 the block of AST shows an example of F2=" ", which means the method name "AssignByteCodeLocations" of the ending statement is not on the list of methods in table 3.

- F3: Elements in collection get updated: The result variable is extracted from the child nodes of the ending statement in the parse tree. The "expression" child node of the loop statement contains the "loop variable". Both the result variable and the loop control variable are thus extracted, and matched to each other to find out if they are the same. The value of F3 is set to 1 if both are same, and 0 otherwise. Please refer to figure 12 to locate result variable and loop control variable in an AST.
- F4: Usage of loop control variable in ending statement: If the loop control variable is directly used in the ending statement or not, is already determined while extracting Feature F3. However F4, provides another option, where the loop variable might be used indirectly in the ending statement through data flow inside the loop. In this scenario, all the statements (of type: assignment or solo_method_call) inside the loop are scanned for possible occurrences of the loop variable or it's derived variables. If any of the derived variable occurs in the ending

statement, F4 is set to 2. Please refer to figure 12 to locate loop control variable and the ending statement in an AST.

• F5: Type of loop exit statement :

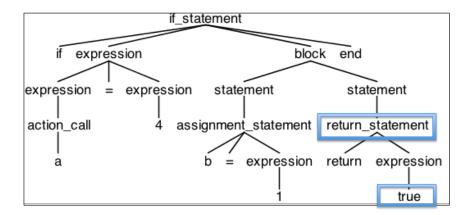


Figure 15: Part of Quorum AST (F5 = 3)

If the last statement inside the "if block" is of type "return statement", the child nodes are further inspected to find out if the statement returns an object or a boolean value. In figure 15 the block of AST shows an example of F5=3, which means the type of loop exit statement is "return boolean".

• F6: Multiple collections in if condition:

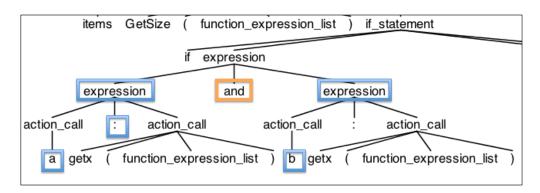


Figure 16: Part of Quorum AST (F6 = 1 and F8 = 2)

The tool extracts the child node "expression" from the "if_statement" node, and splits it around the ":" operator to find the object names used in the statement. F6 is set to 1, if more than one containers are used in the "if_statement". In figure 16 the block of AST shows an example of F6=1, which means there are multiple collections (a and b) in the "if condition".

- F7: Result variable used in if condition: The "result variable" is already been extracted
 to determine feature F3. The child node "expression" of the "if_statement" is again parsed to
 determine F7. Please refer to figure 12 to locate result variable and the "if-condition" in an
 AST.
- F8: Type of if condition: Identifying the operators used in the "if_statement" involves parsing the child node of "expression" (which is a child node to the "if_statement"). The intuition for F8 is for finding max/min element in an array/collection. Having multiple clauses in a if condition is not what is required in this scenario. Hence, if more than one conditions are used in the "if condition", F8 is simply set to 2. In figure 16 the block of AST shows an example of F8=2, which means there are multiple conditions (shown by "and") in the "if condition".

After the feature vectors are extracted, they are mapped to the action identification model in [1], and the high level actions for each of the loop-ifs are determined.

4 Analysis of the Generality of The Loop Action Model

4.1 Applying the Loop Action Model for Quorum

Based on the successful implementation of this project, it can be concluded that applying the Loop Action Model [1] (originally designed for Java) to any other programming language, especially an object-oriented language, is feasible. Identifying a loop-if structure in another procedural programming language (like C, C++, C#, Perl, Python, FORTRAN, MATLAB etc) is achievable, as every procedural programming language has conditional statements (if-statements) and iterative statements (loops). However it would be difficult in pure functional languages (like Haskell, etc) or machine/assembly level languages.

Identifying the relevant pieces like loop control variable, ending statement of "if block", etc are extremely important for determining the feature vector of a "loop-if". However the complexity of identifying those largely depends on the structure and type of the loop and conditional constructs of the language. A for-loop statement is available in most imperative programming languages. Generally, for-loops fall into one of the following categories: Traditional for-loops (e.g. Java, C++), Iterator-based for-loops (e.g. Python), Vectorised for-loops (e.g. FORTRAN 95) and Compound for-loops (e.g. ALGOL 68) [5]. While identifying the relevant pieces for vectorised for-loops would be difficult, it might be straight-forward for the rest. The conditional constructs is mostly common across many programming languages. Although the syntax varies quite a bit from language to language, the basic structure remains the same [6].

The possible feature values of a "loop-if" structure were designed by Wang et al. keeping in mind the characteristics of Java loop-ifs only [1]. So some of the feature values are inapplicable to loop-ifs of other languages. For example, the feature F5 has a possible value of 5, which means the type of the loop exit statement is "throw". But, Quorum does not support the keyword "throw", and hence F5 = 5 is not a possible value in this case. Similarly, feature F1 has a possible value of 5,

which means the type of the ending statement is "object method invocation". Evidently, this feature value would only be possible for object-oriented languages. Also, the set of method names used as the possible values of feature F2 might not exist for other languages. However, in general, it can be concluded that feature extraction would be easier for object-oriented languages, compared to the rest.

4.2 Evaluation of Automatically Identified Actions in Quorum

4.2.1 Methodology

To determine the potential impact of automatically identifying the high level actions of loop-ifs, we ran the action identifier on all Quorum language source code repository that was used as our test data set. Cumulatively, there are 468 programs for the compiler and more than thousand for library files. Data was gathered on the frequency of each high level action that was automatically identified.

4.2.2 Results

In the data set, 40 loop-ifs were identified in total, out of which, high level actions were identified for 20 loop-ifs (i.e. 50%), after mapping the feature vectors with the existent action identification model. 4 types of high level actions were identified, in the frequency of: 'max/min'-(3), 'find'-(4), 'get'-(5), 'determine'-(8) as depicted in Figure 17.

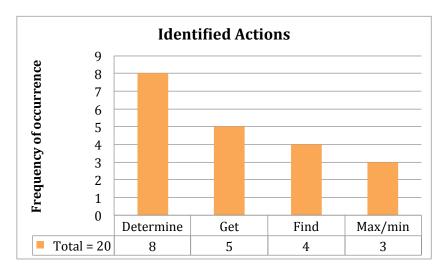


Figure 17: Identified high level action distribution in Quorum Source Repository

As the source code repository (compiler and standard library files) of the Quorum programming language was the only data set available for validation, measuring the correctness of the results of the automatic action identifier was straight forward. The feature vectors and the actions identified by the automatic action identifier for the 20 loop-ifs (for which the high level actions are identified) were matched against the results of the Manual Action Identification (discussed in section 3.2) of

the same loop-ifs. It was found that no incorrect action is identified for any of the 20 loop-ifs, which gives a 100% accuracy of the action identifier tool for Quorum.

For the 20 loop-ifs which could not be identified, 11 were not identified as no corresponding entries were found in the action identification model, and for the rest, actions could not be identified as the last lexical statement of the 'if block' contained method calls with method names not listed in the action identification model used. This is expected as the action identification model [1] was based on top 100 most frequent feature vectors in Java, the table does not have entries for all possible combination of feature value pairs.

Out of the 11 loop-ifs which could not be identified (for no corresponding entries) - a few were very close to identifying action 'find'. To appreciate the problem of exploring whether the loop action model can be modified for Quorum, consider the following example Quorum loop code segment:

```
action GetStaticKey returns text
   key = ""
   i = 0
   repeat GetSize() times
   key = key + names:Get(i)
   if i < GetSize() - 1
        key = key + "."
   end
   i = i + 1
   end
   return key
end</pre>
```

The feature vector for the example code fragment is: (F1:1, F2:0, F3:0, F4:2, **F5:0**, F6:0, F7:0, F8:1). F1 indicates that the ending statement is an assignment. F2 indicates there is no method name for an ending statement method call. F3 indicates that no element in a collection is updated. F4 indicates that the loop control variable is indirectly used through the data flow in the ending statement. F5 indicates that there is no loop exit statement. F6 indicates that there is no collection in if condition. F7 indicates that the result variable is not used in the if condition. F8 indicates that the type of the if condition is not numeric comparison. Mapping the feature vector for this sample Quorum code with the action identification model in table 2, loop action for the particular loop cannot be identified because no corresponding entry was found in the action identification model. However, the feature vectors in table 2 for identifying action find are (F1:1, F2:, F3:, F4:1,2, F5:1,2,4, F6:, F7:, F8:) and (F1:0, F2:, F3:, F4:, F5:4, F6:, F7:, F8:). It is noted that the feature vector for this sample Quorum code is very close to the first feature vector for the action find, except that F5 is 0, not 1,2 or 4. So as future work, there is a potential of modifying the action identification table according to the specifics of Quorum language, for better effectiveness.

4.2.3 Comparing the results: Java vs. Quorum

Keeping in mind the small size of data set currently available for research to implement the model for Quorum, the results look quite promising when compared to Java. The percentage of automatically identified high level actions in both the languages is close to 50%, and accuracy of the model implemented in Quorum is 95%, which seems to be better than what we had for Java which was determined through non-author human judgements. The number of types of identified loop actions in Quorum is low, which might be accounted to the fact that the model was designed keeping in mind the most frequent loop actions occurring in Java. The top 3 frequently identified loop actions are still the same for both the programming languages, only in a different order.

Category	Java	Quorum	
Data set	7,159 open source projects	468 compiler source code programs	
Number of identified loop-ifs	337,294	40	
Number of automatically identified high level actions	195,277 (i.e., 57.9%)	20 (i.e., 50%)	
Types of identified high level actions	12	4	
Accuracy	93.9%	100%	
Most Frequent identified loop actions (in descending order)	Find, Determine, Get	Determine, Get, Find	

Table 4: Comparing Results: Java (left) and Quorum(right)

4.3 Threats to Validity

The results obtained for the Quorum language source code might differ when tested on larger data sets. Also, given the code is written by a handful of developers, there is less variety of loop implementations to test, as different programmers have different coding styles. To mitigate this, as many as available Quorum loops were collected for this project. These loops are real code examples used in writing the Quorum compiler and libraries, and are not some random sample code snippets.

5 Related Work

The first general and extensible approach to automatically identifying action units and abstracting them as high level action phrases without manually creating templates was developed by Wang et al. [1]. The closest work is by Sridhara et al. who automatically generated high-level actions within methods [14], by using a small set of templates that were developed by manually examining code.

This project is related to generating internal comments for the identified high level actions. Most comment generation work is focused on creating summaries for methods or classes [12] [13]. However, Wong et al. mine question and answer sites for automatic comment generation [15]. They extract code-description mappings from the question title and text, use heuristics to

refine the descriptions, and use code clone detection to find source code snippets that are almost identical to the code-description mapping.

6 Conclusion and Future Work

Application of the Loop Action Model for Quorum demonstrated the feasibility of implementing the same model to other programming languages apart from Java. Building a tool for automatic identification of high level loop actions shows the potential to generate internal comments for loop structures and thus help programmers to save time and effort in apprehending code. Also, beyond an approach to generating internal comments, tools that rely on the words in the source code and comments for analysis will benefit when the high level action words do not already appear in the loop code (e.g., search tools, comment generator tools).

Based on the success of this approach, it is worth exploring the applicability of the approach to other programming languages apart from Java and Quorum. It will also be interesting to investigate how the automatic tool works on unseen Quorum code written by different programmers, and determine the accuracy and types of high level actions occurring on a larger data set.

References

- [1] Xiaoran Wang, Lori Pollock and K Vijay-Shanker. "Developing a Model of Loop Actions by Mining Loop Characteristics from a Large Code Corpus" In Proceedings of 31st International Conference on Software Maintenance and Evolution, Bremen, Germany, September 2015.
- [2] Terrence Parr "The Definitive ANTLR 4 Reference". Book, Published by *The Pragmatic Bookshelf*, copyright 2012, The Pragmatic Programmers, LLC, ISBN-13:978-1-93435-699-9
- [3] Quorum Source Code Repository, https://bitbucket.org/stefika/quorum-language
- [4] Quorum Programming Language, http://quorumlanguage.com/
- [5] Types of For Loops, https://en.wikipedia.org/wiki/For_loop
- [6] Types of Conditional Constructs, https://en.wikipedia.org/wiki/Conditional_ %28computer_programming%29
- [7] A. M. Stefik, C. Hundhausen, and D. Smith, "On the design of an educational infrastructure for the blind and visually impaired in computer science" in *in Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education, SIGCSE 11.* New York, NY, USA: ACM, 2011, pp. 571?576.
- [8] ANTLR4. ANTLR4, January 2013.
- [9] Alfred V Aho et al. Compilers: principles, techniques, & tools. Pearson Education India, 2007.
- [10] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance" in 23rd annual Intl Conf on Design of communication, 2005. New York, NY, USA: ACM, 2005, pp. 68-75.
- [11] M. Kajko-Mattsson, "The state of documentation practice within corrective maintenance" in ICSM ?01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM?01) Washington, DC, USA: IEEE Computer Society, 2001, p. 354.
- [12] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.
- [13] L.Moreno, J.Aponte, G.Sridhara, A.Marcus, L.Pollock, and K Vijay-Shanker Automatic generation of natural language summaries for Java classes. In *Program Comprehension (ICPC)*, 2013 IEEE 21st International Conference.
- [14] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 101–110. IEEE, 2011.

- [15] E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE)*, 2013 IEEE/ACM 28th International Conference on, pages 562–567, Nov 2013.
- [16] R.P.L. Buse and W. Weimer. Synthesizing api usage examples. In *Software Engineering* (ICSE), 2012 34th International Conference on, pages 782–792, June 2012.
- [17] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE)*, 2010 17th Working Conference on, pages 35–44, 2010.
- [18] Letha Etzkorn, Carl Davis., and Lisa Bowen. The language of comments in computer software: A sublanguage of English. *Journal of Pragmatics*, 33:1731–1756(26), November 2001.
- [19] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers Taxonomies and characteristics of comments in operating system code. In 31st Intl Conf on Software Engineering (ICSE09), May 2009.
- [20] P.-N. Robillard. Automating comments. SIGPLAN Not., 24(5):66–70, 1989.
 Nov 2000.
- [21] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *J. Prog. Lang.*, 4(3), 1996.
- [22] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 145–158, New York, NY, USA, 2007. ACM.
- [23] T. Tenny. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.*, 14(9):1271–1279, 1988.
- [24] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Reverse Engineering (WCRE)*, 2011 18th Working Conference on, pages 35–44. IEEE, 2011.