Java loops are mainly polynomial*

Maciej Zielenkiewicz, Jacek Chrząszcz, and Aleksy Schubert

Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Abstract. Although there exists rare cases where exponential algorithms are used with success, practical software projects mostly consist of polynomial code. We present an automatic analysis tool which divides while-loops in a Java software project into clearly polynomial ones and the rest. The analysis can be useful for example in software quality assurance, maintenance and design of new programming language idioms. After running our tool on two sets of several medium size Java projects we conclude that almost 80% of while-loops are trivially polynomial.

1 Introduction

Most work on improving existing programming languages, coding habits or program development methodologies are based on problems whose existence comes from common knowledge and does not result from any thorough analysis of existing code. In this work we propose an automatic analysis of while-loops. Our goal is to detect the loops which, in intent, run in polynomial time. This complexity is often identified with the limit of efficient computation, although a number of practical exponential algorithms are in everyday use (e.g. extended regexp matching, see [5]). We establish a number of typical cases, such as counting up to a specified integer, reading an input stream, or iterating over a data structure. In the end it turns out that about 80% of the while-loops are trivially polynomial. Our analysis is rough, it is not backed up by any formal treatment of the loops' complexity. Its goal is to provide an experimental background for a proposal of a new programming language in which polynomial loops would be syntactically or statically distinguished from the others. Our aim is an evolutionary change in programmers' habits with a view towards more readable and more maintainable code. In particular we want to be able to statically detect more programming errors such as unintended change of loop control variable. The proposal itself is out of scope of the current paper.

It can be reasonably expected that most loops should be polynomial and the others likely have some kind of mistake which affects their termination. In fact it is rare to find correct code which is not polynomial in a regular software project. This work serves as a quantitative base of this assumption; we propose a number of categories of polynomial loops, encode them as XPath expressions over abstract syntax trees and using a code scanning tool CodeStatistics [4], for

* This paper is partially supported by Polish government grant N N206 493138.

a given Java project we can calculate the number of loops that are easily checked as being polynomial.

Our analysis is not a precise one. Unlike the line of research connected with Implicit Complexity Classes [2], we do not aim at precise linguistic description of the polynomial complexity class. Instead, we work on real code and give only an estimate on the relative number of polynomial loops. Of course there are many loops which are perfectly correct (and polynomial) and do not fit into any of our categories. On the other hand, the number of iterations of some loops categorized as polynomial may be bounded by a variable's value, which itself may be exponential with respect to the size of input. Nevertheless we treat them as polynomial.

A similar non-strict analysis of code is done by many tools such as FindBugs [8] or CheckStyle [1], which help limit the quality assurance effort without providing any formal guarantees. Using these programs as part of regular development toolchain can greatly improve the code quality without inducing any significant cost. Our analysis can also serve this purpose, but is rather dedicated to senior programmers, whose role includes reviewing the work by others.

A few static approaches to inference of complexity of loops were described previously. One of them was proposed by Shkaravska, Kersten, and van Eekelen [10] and is based on assumption that running time is given exactly by a polynomial (or combination of polynomials depending on the loop condition), coefficients of which are determined with test runs of the loop. The other approaches are generation of bounds which are linear combinations of variables, developed by Gulwani [6], or derivation of numerical upper bounds of the execution time of the program, presented by Ermedahl et al [3].

The paper is organized as follows. Section 2 presents the details of the different loop categories together with examples of corresponding XPath rules. Section 3 reports the results of the study performed on a set of several open source Java projects together with actual code snippets presenting various loop categories. The numbers resulting from this analysis are compared with the analysis of another set of Java projects. In Section 4 we present related work on for-loops that can easily be combined with our analysis and we conclude in Section 5.

2 Analysis

2.1 Cost model

Analysis of complexity is always relative to a cost model associated with the computation process. The cost model should describe (1) which data is taken into account as input, (2) which resources are counted (e.g. time, memory, network connections etc.) and (3) which elements of the computation model dynamics cause charging for the resource (e.g. firing of a rule in a Turing machine is charged one time unit).

In this study, we focus on running time analysis of Java programs. The input we take into account are the numerical variables used in the loop as well as the

graph structure of the objects reachable from the variables in the same range. We charge one time unit for a simple operation such as assignment, addition, control-flow split etc.

In general, there are three approaches possible for measuring complexity of programs operating on numerical variables. The first two are strict from the complexity theory point of view, but highly counter-intuitive. The third one is a compromise, which follows the human intuition about program complexity, but has some strange properties as far as the complexity theory is concerned.

The first approach consists in viewing all 32-bit integers as symbols of a Turing machine alphabet rather than actual input data. In this view, input data is stored in external massive storage devices such as disks since this may make a potentially infinite input size (see e.g. [9]). This model follows our intuition when operations on data-structures are considered, but is rather disconcerting when numeric programs are considered. Indeed, basic operations on integers are naturally constant-time operations, as they produce one integer as a result. But it follows that all numeric programs run basically in constant time, as they use at most constant number of additional 32-bit integer variables, which all can range over a constant number of values, so the total number of configurations is constant (although this might sometimes be quite a big constant).

The second model treats machine integers as arbitrary large bit sequences. In this view basic operations on integers, such as addition, have linear complexity, and any program whose number of operations is linear with respect to the number represented by the input value is indeed exponential, which is again contrary to the intuition. This model is followed quite often when cryptographic algorithms are analysed.

The model that is most often adopted in the analysis of algorithms and programs consists in treating integers as unary values, e.g. an integer of value n has size n, but nevertheless considering basic operations on integers as atomic, i.e., done in constant time. This model is appropriate to the human intuition about complexity, but leads to paradoxes as far as complexity theory is concerned. For example consider the program in Figure 1. It works in linear time with respect to

```
k = 1;
i = 0;
while (i < n) {
  k = k + k;
  i++;
}
```

Fig. 1. Linear program calculating an exponential value

the value (and size) of n, but the final value of k is 2^n, whose size is exponential. So it produces exponential output in linear time. No Turing Machine can do such a thing!

The view we adopt in this paper is the usual one. It is important to remark that our meaning of polynomial loops amounts only to limiting the number of iterations of the loop. Such polynomial loops do not compose well, in a sense that nested or even subsequent polynomial loops can in fact run in exponential time. Consider the two programs presented in Figure 2 (loops are presented as for-loops for the sake of readability).

```
k = 1;
for (i = 0; i < n; i++) {
  k = k + k;
  for (j = 0; j < k; j++) {
    doSomething();
  }
}
```

```
k = 1;
for (i = 0; i < n; i++) {
  k = k + k;
}
for (j = 0; j < k; j++) {
  doSomething();
}
```

Fig. 2. Nested polynomial loops make exponential program.

Although in both programs the number of iterations of both loops is bounded by values of variables which are not modified in the loop body, since the value of k can grow to be exponential, doSomething is executed exponential times as so is bounded the number of iterations of the j-loop. At the same time the j-loop analysed in isolation from the rest of the code is perfectly polynomial since given a bound k it iterates polynomial times with respect to the value.

Having in mind these paradoxes we consciously decided to follow this inconsistent model in our study. The main reason for this is that we want to obtain a picture based upon the way the programmers view their code. We believe also that the community of programmers is likely to adopt solutions that help in code management in accordance with their point of view and they will disregard methods that claim a program is non-polynomial while they see at first sight it is such.

2.2 Scope

We will be analysing while-loops in sources of six open-source Java programs: Apache Hadoop, Google App Engine SDK, JEdit, Hibernate, Oracle Berkeley DB, Apache Tomcat (for exact versions see Table 3. This is the same selection as of the previous analysis done on the for-loops with CodeStatistics, which covered the for-loops [4]. The body of the programs outside of the loops was not analysed and recursion is not taken into account.

In the chosen projects almost 35% of loops were while-loops, while the other approximately 65% were for-loops. We do not consider do-while-loops as they were less then 1% of the total number of loops. We do not analyse the for-loops because a similar analysis was done previously be Fulara and Jakubczyk [4], but we do describe how their categories interact with our notion of polynomial loop.

Fig. 3. Versions of programs used as sources for experiments. The second set of programs was used for verification of rules.

Program	Version
Apache Tomcat	7.0.32
Googla App Engine SDK	1.7.2.1
Apache Hadoop	1.0.4
Hibertate	4.1.7.Final
Oracle Berkeley DB	5.0.58
JEdit	5.0pre1
AspectJ	1.7.2
Spring Framework	3.1.1
Vuze	5.0.0.1

For the sake of simplicity we will not be analysing termination of loops by exception-throwing, as it is generally hard in Java provided the existence of unchecked exceptions, but can only lead to a decrease of number of computations.

The aim of this work is to classify at least 80% of the loops, which is means most of loops could be easily spelled out with specific constructions in a programming language which would utilize specific loops for different levels of complexity. The remaining 20% could be marked as unsafe and left for more detailed checks by the programmer.

2.3 Tools

The tool selected to be used for analysis of Java code was CodeStatistics [4], which translates abstract syntax tree of a Java source file to XML and is capable of making statistics of numbers of matches for specified XPath expressions. For more information it can be requested that the matched fragments are output together with the expression which matched them. An example of a rule is presented in Figure 4.

```
<description name="while-read-write" xpath='//WhileStatement[
(
    //MethodInvocation[starts-with(@methodName, "read")]
    or
    //MethodInvocation[starts-with(@methodName, "write")]
    or
    //MethodInvocation/Name/attribute::fullname = "System.in"
)
]'/>
```

Fig. 4. An example of a rule definition. This rule selects all loops which either call method *read* or *write* on any object or call any method of *System.in*.

2.4 General levels of classification

In the high-level view there are a few classes of loops for which the assignment of computational complexity needs further clarification. We will also present in this section a few general remarks on the assumptions made on the usual problems with static analysis of Java programs.

System interaction There are loops for which main reason of existence is interaction with the outside world. This includes for example reading or writing data, waiting for other threads and so on. We will classify these loops in a separate class as the complexity of the interaction is generally not defined in the usual computation models such as Turing machines.

Polynomial loops The main goal of the analysis is to identify the loops with polynomially bounded time. It was chosen to consider a loop as polynomially-bounded if its execution time is bound by a single variable polynomial, for which the variable is a value which either exists in memory or can be polynomially calculated from values in memory.

Aliasing In all places where this may be a problem potential aliasing of variables is not taken into account, i.e. we consider all objects to be different instances. For more detailed view of the problem see for example [7].

3 Results

3.1 Rules of classification

System interaction: sleep-wait-notify. This rule selects all loops in which a call to *Thread.sleep* or one of the methods *wait*, *notify*, *notifyAll* on any object is done. An example of a loop which is categorized in this category is shown in Figure 5.

```
while (!stopAwait) {
  try {
    Thread.sleep(10000);
  }
 catch (  InterruptedException ex) {
  }
}
```

Fig. 5. An example of rule in class system interaction: sleep-wait-notify.

System interaction: read-write. This rule catches all loops where method *read* or *write* of any object is called or there is a call to any method of *System.in*; such a loop is presented in Figure 6. The calls to method other then *write* on *System.out* are specifically not caught as it turned out that they are mostly used for debugging or logging of messages, which would classify otherwise innocent loops as system interactions.

```
while ((line=br.readLine()) != null) {
  if (version == null && line.contains(key)) {
    version=line.substring(line.indexOf(key) + key.length(),
        line.length() - 1);
  }
}
```

Fig. 6. An example of rule in class system interaction: read-write.

All the following classes include loops that satisfy the stated condition but do not belong to any of the "system interaction" classes.

Linear iteration: iterators. This category catches all loops where the condition either is just an invocation of a method with one of the names *hasNext*, *hasPrevious*, *hasMoreElements*, *hasMoreTokens*, *isEmpty* on an object or an expression which includes invocation of method *size* or *length*, for example like in Figure 7.

This should catch most of the usual iterations with standard collections' iterators, the *Enumerable* interface and usage of *StringTokenizer*. As in general it is not possible to tell if a collection is not altered in loop body, 100 loops matching this rule were selected at random and hand-checked for additions to the iterated collection, and it was found out that there were none. It was found out as well that 99% of checked loops had a *next* (or corresponding) call on each if-branch.

Linear iteration: local counter. The existence of this rule is motivated by the following example loop:

```
while (end > start) {
  byte temp=buf[start];
  buf[start]=buf[end];
  buf[end]=temp;
  start++;
  end--;
}
```

Assuming non-negative starting values and no overruns the condition will eventually fail, as the direction of increasing/decreasing of variables is towards the check failing. The example was generalised to select all loops in which at

```
while (i < referenceCount) {
  int source=srcAndRefPositions[i++];
  int reference=srcAndRefPositions[i++]; int offset;
  if (source >= 0) {  offset=position - source;
    if (offset < Short.MIN_VALUE ||
        offset > Short.MAX_VALUE) {
      int opcode=data[reference - 1] & 0xFF;
      if (opcode <= Opcodes.JSR) {
        data[reference - 1]=(byte)(opcode + 49);
      } else { data[reference - 1]=(byte)(opcode + 20); }
      needUpdate=true;
    }
    data[reference++]=(byte)(offset >>> 8);
    data[reference]=(byte)offset;
  } else {
    offset=position + source + 1;
    data[reference++]=(byte)(offset >>> 24);
    data[reference++]=(byte)(offset >>> 16);
    data[reference++]=(byte)(offset >>> 8);
    data[reference]=(byte)offset;
  }
}
```

Fig. 7. An example of rule in class linear iteration: iterators.

least one of the check operands is a local variable that is increased or decreased in the desirable way (only operators `--` and `++` are taken into account), is not assigned to, and if the other operand is a local variable as well it is not assigned to and there are not increases/decreases in the undesirable way.

3.2 Classification of loops

The results of classification are given in Figure 8. It should be noted that the intersections between categories are not always empty (but the "linear" classes explicitly exclude the "system interaction" classes; the total number of 80.87% of loops which are categorized is the difference between 100% and the number of uncategorized loops, so that it is correct with non-zero intersections as well.

To make the intersections more clear additional calculation of the number of loops which are categorized simultaneously in two categories was performed. Results of that analysis are shown in the Figure 10. The rules which excluded system interactions from other classes were turned off for that analysis, therefore numbers for some classes are bigger than in Figure 8.

After developing the set of rules a second evaluation was done on additional set of programs in order to check if the rules were not overtrained to specific examples used during test. The results are shown in Figure 9; in total 72% of loops were categorized.

program	all loops	uncategorized	linear iteration		system interaction	
			iterators	local counter	read write	sleep-wait notify
appeng	6	2	1			3
hadoop	1274	137	82	47	890	472
hibernate	739	94	464	8	170	5
berkdb	830	241	123	42	319	175
jedit	394	143	65	36	138	12
tomcat	1017	198	219	65	461	187
total	4260	815	954	198	1978	854
percent of all		19.13%	22.39%	4.65%	46.43%	20.05%
categorized loops			**80.87**%			

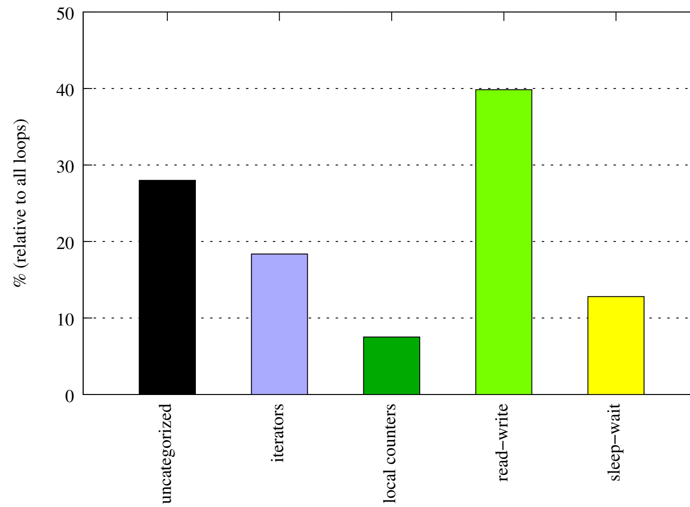

Fig. 8. Results of categoriaztion of while-loops with rules given in Section 3.1.

3.3 Sightings

During the experiments a number of interesting sights could be noticed, some of which will be described in this section.

Interesting loops. The loops which were not categorized in any category include some for which the complexity, or, more generally, the way of terminating, is not seen at the first glance, and therefore they should be commented and given more attention by the programmer. Consider the following loop as an example:

program	all loops	uncategorized	linear iteration		system interaction	
			iterators	local counter	read write	sleep-wait notify
aspectj	1453	626	93	176	532	74
spring	187	98	55	8	9	8
vuze	1485	151	426	51	704	318
total	3125	875	574	235	1245	400
percent of all		28.00%	18.37%	7.52%	39.84%	12.80%
categorized loops			**72.00**%			

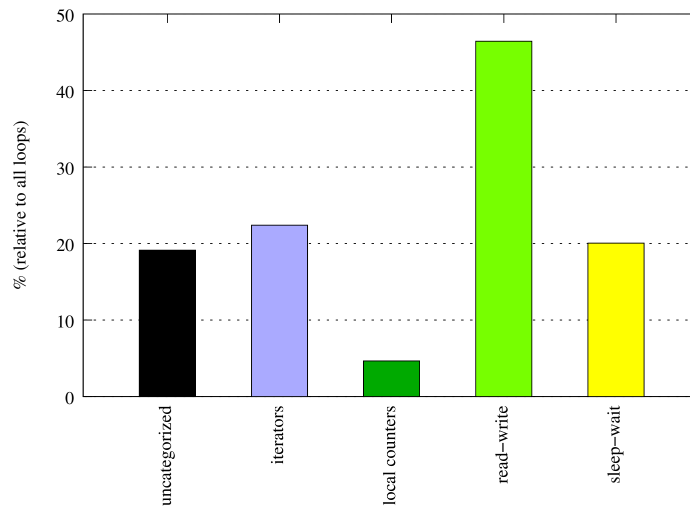

Fig. 9. Results of classification of while-loops for second set of test programs.

Fig. 10. Intersections of classes as absolute numbers and percentages of their classes.

	iterators	local counter	read-write	sleep-wait-notify
iterators	0	12	1050	166
local counter		0	230	117
read-write			0	532
sleep-wait-notify			0	

		iterators	local counter	read-write	sleep-wait-notify
		1537	478	1978	854
iterators	1537	0	3% / 1%	53% / 68%	19% / 11%
local counter	478		0	12% / 48%	14% / 24%
read-write	1978			0	62% / 27%
sleep-wait-notify	854				0

```
1  while (current != null && !mappings.
       getReflectionManager().toXClass(Object.class).
       equals(current)) {
2    if (current.isAnnotationPresent(Entity.class) ||
         current.isAnnotationPresent(MappedSuperclass.
         class) || current.isAnnotationPresent(Embeddable.
         class)) {
3      Map<String,Column[]> currentOverride=
           buildColumnOverride(current,getPath());
4      Map<String,JoinColumn[]> currentJoinOverride=
           buildJoinColumnOverride(current,getPath());
5      Map<String,JoinTable> currentJoinTableOverride=
           buildJoinTableOverride(current,getPath());
6      currentOverride.putAll(columnOverride);
7      currentJoinOverride.putAll(joinColumnOverride);
8      currentJoinTableOverride.putAll(joinTableOverride)
           ;
9      columnOverride=currentOverride;
10     joinColumnOverride=currentJoinOverride;
11     joinTableOverride=currentJoinTableOverride;
12   }
13   current=current.getSuperclass();
14 }
```

After a short analysis it can be noticed that the loop is iterating upward the tree of inheritance and is always terminating due to absence of loops in the inheritance tree.

Another example is the following code:

```
1  while (!Thread.currentThread().isInterrupted()) {
2    try {
3      final JobStory job=getNextJobFiltered();
4      if (null == job) {
5        return;
6      }
7      if (first < 0) {
8        first=job.getSubmissionTime();
9      }
10     final long current=job.getSubmissionTime();
11     if (current < last) {
12       LOG.warn("Job␣" + job.getJobID() + "␣out␣of␣
             order");
13       continue;
14     }
15     last=current;
16     submitter.add(jobCreator.createGridmixJob(conf,
           initTime + Math.round(rateFactor * (current −
```

```
        first)),job,scratch,userResolver.getTargetUgi(
        UserGroupInformation.createRemoteUser(job.
        getUser())),sequence.getAndIncrement()));
17    }
18    catch (  IOException e) {
19       error=e;
20       return;
21    }
22 }
```

The termination of the loop depends on receiving an interrupt from another thread (or an exception), whereas otherwise it is an "sincerely infinite" loop which processes jobs as long as there are any left.

while(true) loops. Some of the loops have the condition *true*. This is not surprising, as they can be exited in other ways, for example by *return*, *break* or *return*. But in 0.28% of all loops the condition is *true*, but none of the aforementioned instructions is found. These are the loops which certainly should require more programmer attention and a probably a comment of explanation (one way of returning from such a loop could be for example by an exception thrown from a function called inside the loop body).

4 Linearity of for-loops

After categorizing over 80% of while-loops it is a natural extension to check if a similar result can be obtained for for-loops. Building onto work of Fulara and Jakubczyk [4] let's recall their result of generating *decreases* formula for 74.4% of for-loops. Considering their categories and how the expression which shall decrease is built we can see what is it's relationship to other variables in program, and therefore infer an upper bound on the complexity. All of the rules used in the referenced results excluded any additional modification to control variables; that means that it is necessary only check the other operand of the comparison in loop guard. It turn out that in every case this is a known upon entering the loop:

Literal. A literal cannot be changed in any way, so literal-bounded loop has complexity $\mathcal{O}(1)$.

Constant. Using a constant is practically equivalent to using a literal (with a type), so the complexity is still $\mathcal{O}(1)$.

Final field. A final field in a program is effectively a constant known upon entering the loop.

Local expression. This class consists of loops for which the number of loop body executions is bounded by an expression which uses only local variables which are not modified in loop body. The complexity of this loop depends on the expression which is used: it is polynomial if the expression is polynomial. As we do not check the functions which may be called the only way to write an expression which is not polynomial in our case is to use bit shift or unary negation operators, which can produce numbers which are exponential functions of their operands. Additional check was ran for that category to check if the expressions contain aforementioned operators, and it was found out that there are no uses of those operators.

Taking above analysis of for-loops into account it turns out that all of the previously categorized for-loops have polynomial complexity. That meas than in total we have

$$74.4\% \cdot 65\% + 80.8\% \cdot 35\% \approx 76.6\%$$

of loops categorized.

5 Conclusions and future work

We have presented an automatic analysis tool which detects clearly polynomial while-loops in a Java software project. The polynomial loops are divided into several categories, such as iterating to a specified integer value, reading an input stream, or visiting a data structure. We ran our tool on two sets of several medium size open source Java projects and as expected it turned out that almost 80% of while-loops are trivially polynomial.

The analysis we presented can be extended in various ways. One of the extensions would be to transform the categories of while-loops into the corresponding ones for the for-loops, which would improve the analysis described in [4].

The presented analysis constitutes a quantitative basis for further investigations on how to apply implicit computational complexity methods to real languages such as Java.

References

1. Checkstyle, `http://checkstyle.sourceforge.net/`
2. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. computational complexity 2(2), 97–110 (1992), `http://dx.doi.org/10.1007/BF01201998`
3. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: Rochange, C. (ed.) 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2007), `http://drops.dagstuhl.de/opus/volltexte/2007/1194`

4. Fulara, J., Jakubczyk, K.: Practically applicable formal methods. In: Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science. pp. 407–418. SOFSEM '10, Springer-Verlag (2010), http://dx.doi.org/10.1007/978-3-642-11266-9_34
5. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
6. Gulwani, S.: Speed: Symbolic complexity bound analysis. In: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 51–62. CAV '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02658-4_7
7. Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R.: The geneva convention on the treatment of object aliasing. SIGPLAN OOPS Mess. 3(2), 11–16 (Apr 1992), http://doi.acm.org/10.1145/130943.130947
8. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Notices 39(12), 92–106 (2004)
9. Kosovskiy, N.K.: Polynomial-time program conditions for three programming languages (2013)
10. Shkaravska, O., Kersten, R., van Eekelen, M.: Test-based inference of polynomial loop-bound functions. In: Krall, A., Mössenböck, H. (eds.) PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. pp. 99–108. ACM Digital Proceedings Series (2010)