Modeling Time in Java Programs for Automatic Error Detection

Giovanni Liva
Alpen-Adria-Universität
Klagenfurt, Austria
giovanni.liva@aau.at

Muhammad Taimoor Khan
Alpen-Adria-Universität
Klagenfurt, Austria
muhammad.khan@aau.at

Francesco Spegni
Universitá Politecnica delle Marche
Ancona, Italy
f.spegni@dii.univpm.it

Luca Spalazzi
Universitá Politecnica delle Marche
Ancona, Italy
l.spalazzi@dii.univpm.it

Andreas Bollin
Alpen-Adria-Universität
Klagenfurt, Austria
andreas.bollin@aau.at

Martin Pinzger
Alpen-Adria-Universität
Klagenfurt, Austria
martin.pinzger@aau.at

ABSTRACT

Modern programming languages, such as Java, represent time as integer variables, called timestamps. Timestamps allow developers to tacitly model incorrect time values resulting in a program failure because any negative value or every positive value is not necessarily a valid time representation. Current approaches to automatically detect errors in programs, such as Randoop and FindBugs, cannot detect such errors because they treat timestamps as normal integer variables and test them with random values verifying if the program throws an exception. In this paper, we present an approach that considers the time semantics of the Java language to systematically detect time related errors in Java programs. With the formal time semantics, our approach determines which integer variables handle time and which statements use or alter their values. Based on this information, it translates these statements into an SMT model that is passed to an SMT solver. The solver formally verifies the correctness of the model and reports the violations of time properties in that program. For the evaluation, we have implemented our approach in a prototype tool and applied it to the source code of 20 Java open source projects. The results show that our approach is scalable and it is capable of detecting time errors precisely enough allowing its usability in real-world applications.

CCS CONCEPTS

• **Software and its engineering** → **Error handling and recovery**; • **General and reference** → *Verification*; • **Theory of computation** → *Program semantics*; Abstraction;

1 INTRODUCTION

The majority of software failures are predictable and avoidable [8]. Typically, errors are detected through testing the software implementation. When the final version of the implementation passes all tests, it could be marked as an accepted implementation of the requirements. However, writing tests that cover all possible scenarios in which the application is used, is impossible. Many tools have been developed to aid developers to find and correct bugs before they are released. For example FindBugs [17] performs a static analysis of the source code and based on some predefined syntactical patterns, provides hints to the developers about locations of the code that could suffer from errors. Modern approaches, such as Randoop [19–22] and Agitator [6], can create automatically unit tests for a project to stress the application with the purpose to discover errors and create a regression test suite. The tests it generates are a random sequence of method and constructor invocations for the class under test.

Identifying bugs in early stages of the development cycle is challenging, even more the one related to time. Modern programming languages, such as Java and C#, allow developers to model time as timestamps using integer variables. However, all of the aforementioned approaches fail to identify errors that are related to the usage of time. Those approaches test methods with some random integer values for the parameters and verify if they throw a runtime exception. For instance, an integer related exception can be thrown for an illegal array index access (*IndexOutOfBoundsException*), when a number is divided by zero (*ArithmeticException*) or when it is malformed (*NumberFormatException*). Therefore, they miss to identify time related errors because no runtime exception is thrown as the chosen parameters are legal integer values. In the timestamp domain, only non-negative values are acceptable. Positive values are more subtle because they represent a valid and legal value for timestamps in principle, however operation between them could result in an error due to an integer overflow.

Listing 1 shows an example taken from a bug of the Apache Kafka project. Depending on the value of the input parameter, the `deadline` variable, which stores a timestamp value, in line 3 can be assigned an illegal value. Since the parameter is defined as `long`, some input values for the parameters will go un-noticed by the tests but will result in a program failure at runtime. In fact, `deadline` can be lower than `now` and this in turn prevents the body of the while loop to be executed. This case can cause a failure in the program because developers expect the `deadline` variable to

always be greater than now. In the description of the issue, the Kafka developers state that they found the bug out of the box.

In this paper, we present an approach to automatically and systematically verify time related errors in Java programs. With a formal time semantics, we determine which integer variables handle time and which statements use or alter their values. Based on this information, our approach analyzes this subset of the source code and when it encounters a branching instruction it creates a copy of the code to handle every path of the branch in order to obtain a full path coverage. For each copy, it creates a set of constraints that model the time behavior of the program. Finally, it passes the set of constraints as input to the Z3 [2, 11] Satisfiability Modulo Theories (SMT) solver. The SMT solver is the oracle that reports all cases in which time properties cannot be held in the given model. A formal semantics of the Java language was introduced by Bogdanas *et al.* [5]. On top of their work, we presented [18] an extension that specifies the time semantics of the Java 8 language that we use in our approach.

We have implemented our approach in a prototype tool[1] and we performed an empirical study on 20 open source Java projects to assess to which degree we can correctly identify time related errors. On these 20 projects, we verified 939861 methods detecting 146 errors. We manually confirmed that these are due to real bugs in the source code. We also observed 12 errors that are false positives. The results also show that our tool is able to process on average a method in 7.02 ms. Our studies show that our approach is fast and precise allowing its applicability for finding bugs in practice.

The remainder of the paper is organized as follows: Section 2 presents the motivating example taken from a bug of the Apache Kafka project. Section 3 presents the details of our approach for verifying time related errors in Java programs. In Section 4 we evaluate our approach and we discuss results and threats to validity in Section 5. Section 6 gives an overview of related work and we conclude the paper in Section 7.

2 MOTIVATING EXAMPLE

This section presents an example to motivate why a semantic approach is necessary to identify a bigger spectrum of errors, such as related to time. Modern programming languages, for instance Java and C#, offer APIs to handle time related operations, *e.g.*, the `java.time.Clock` class. However, these APIs encode time values as integer values from the domain of $\mathbb{Z}$ and allow developers to explicitly manipulate time as integer, *e.g.*, with calls to `System.currentMillis()`. We focus on this representation because time by meaning of timestamps has a problem with their representation using integer variables. Timestamps allow developers to tacitly model incorrect time values resulting in a program failure since not every value represent a correct time value. In the next paragraphs, we present an example taken from a real world bug, namely KAFKA-4290[2], where the manipulation of timestamps led to a critical error.

Listing 1 shows the source code of the method `poll` that is responsible for the reported issue. If the method is called with (i) any negative number or (ii) a number which is too large, its execution will cause a failure in the program. The source of the

```
1  public void poll(long timeout) {
2    long now = time.milliseconds();
3    long deadline = now + timeout;
4    while (now <= deadline) {
5      if (coordinatorUnknown()) {
6        ensureCoordinatorReady();
7        now = time.milliseconds();
8      }
9      if (needRejoin()) {
10       ensureActiveGroup();
11       now = time.milliseconds();
12     }
13     pollHeartbeat(now);
14     long remaining = Math.max(0,
15         deadline - now);
16     client.poll(Math.min(remaining,
           timeToNextHeartbeat(now)));
17     now = time.milliseconds();
18   }
19 }
```

Listing 1: Source code of the method `poll()` from the class `WorkerCoordinator` that caused the issue KAFKA-4290.

problem is at line 3, where the value for `deadline` is computed by adding together the current time stored in variable `now` and a timeout value stored in the parameter `timeout`. The parameter specifies the maximum amount of time that the method should require to terminate. However, it is a common practice to pass, as parameter, a big enough value to enforce the normal method termination without preemptively forcing it. Albeit a positive big number is a valid parameter value, the sum at line 3 might result in an integer overflow and the JVM neither throws and exception nor gives an error. This operation will store a negative value in the `deadline` variable and make it smaller than the variable `now` before entering the while loop. The same problem arises if a negative value is passed as parameter. The while loop will be never executed contrary to the developers' expectations and this causes a failure in the program.

In the description of this issue, the developers state: *"We hit this case out of the box ..."*. This statement shows a limitation of the testing approach in which the method is tested only for few values. Unfortunately, we cannot write tests that cover all the possible cases and check the correct behavior of a method for all possible parameters. Modern techniques try to automatically create tests that check the behavior of the system in different settings. Randoop [19–22] is a well known example of such a test generation technique. However, it has no knowledge of what is the intended program behavior and therefore, it could only verify if, with some randomly generated values for the input parameters, the method exhibits an exception. Moreover, it considers timestamp values as normal integer values and therefore it verifies that the program is safe according to the integer properties that are not representative for the timestamp domain. This is the gap that we aim to fill with our approach.

[1] https://github.com/rtse-project/automatic-error-detection
[2] https://issues.apache.org/jira/browse/KAFKA-4290

3 EXTRACTING THE TIME MODEL

In this section, we present our approach for detecting time errors in the source code of Java programs. Figure 1 shows an overview of our approach that consists of three steps. In the first step, the source code of each method of a Java program is analyzed to determine the variables that store time and statements that use or alter them. We call these statements a time slice of a method (all the other statements are filtered out). For each time slice, we next create multiple copies of a method based on its branching instructions that we call a **path**. Each path of the method is then translated into an SMT model fed into an SMT solver. The solver verifies the correctness of the program and reports the errors detected in it.

3.1 Time Slice

Based on the formal time semantics of Java [18], in this step we create a slice of the original program composed of only those statements that modify and use time. In our previous work, we presented a formal time semantics for the Java language and a static analysis approach to identify integer variables that hold time values as *time variables* and statements that alter time as *time statements*. Our approach is based on a categorization of Java 8 APIs methods into three different categories:

(1) **Return Time:** The first category comprises methods that return an integer value that represents time, *e.g.*, the static `nanoTime()` method of the `System` class returns the current time in nanoseconds.
(2) **Explicit Timeout:** The second category comprises methods that accept as parameter a time value. An example is the `sleep` method of the `Thread` class in which the parameter specifies the maximum amount of time to suspend the current thread.
(3) **Explicit Wait:** The last category comprises methods that can potentially block the execution of a thread forever. An example of such a method is the `join()` method of the `Thread` class.

In this step, we apply our static analysis approach to detect time variables, method calls that return time, and statements that use or alter time variables. Based on this analysis, we translate the identified time related statements into four basic types of statements that implement time-related behavior in programs. We call this sequence of statements a *time slice*. All the other statements in the method are filtered out. We consider the following four statement types as the main source to introduce time related errors:

(1) **Assignment Statements:** These statements change the value of a time variable and might assign it a value conflicting the time semantics, *e.g.*, any negative value.
(2) **Method Calls:** Methods might be called with an invalid time value and it might result in an incorrect data-flow.
(3) **While Loops:** The guard of a while loop referencing a time variable in its condition might not be satisfied because of an invalid time value and it could result in an unexpected behavior of the loop. In our approach, the various types of loop statements in Java, such as the `for` or `do-while` loops are rewritten into a `while` loop without changing the semantics of the original loop.
(4) **Conditional Statements:** They model the control-flow of a program and distinct paths might result in different values for the time variables. Note, `switch-case` statements are rewritten into a sequence of `if`-statements.

We translate the four basic types of statements into corresponding SMT constraints while preserving the time semantics of the Java language. Thus, every property discovered with the SMT solver is assured to be also valid for the given method of the program.

Listing 2 shows an example of the translation of the method `poll` from Listing 1 into a time slice. The variables `now`, `timeout`, `deadline`, and `remaining` are marked to be time variables and all statements referencing them are considered time statements. In contrast, the method calls `coordinatorUnknown()` and `needRejoin()` inside the two if conditions and the method calls to `ensureCoordinatorReady()` and `ensureActiveGroup()` are filtered since they do not reference nor return a time variable. Note, since the bodies of both if conditions contain a time statement, they are contained by the time slice.

3.2 Path Generation

This step of our approach creates multiple versions of the extracted time slice that represent the different execution paths of the method. The time slice created could have a nondeterministic behavior for branching statements. This can happen if their conditions are not time related, *e.g.*, line 4–6 in Listing 2. Therefore, it is necessary to perform a linearization of the program considering all possible branches to make it deterministic and translate it into a set of SMT constraints.

We parse the time slice generated in the previous step into a control flow graph. Then, for each possible execution path in the control flow graph that contains a branching statement whose condition is empty (because it is not time related), we create two copies of the program, each is called a **path** of the program: one in which the condition is evaluated to true and one in which it is evaluated to false. In this manner, we have a full path coverage.

Referring to the example given in Listing 2, this step creates the four different Paths *a, b, c,* and *d* of the program presented in Figure 2. The first version, Path a, of the program does not contain the statements of the bodies of both if statements since both are evaluated to false. The second version, Path b, contains the statement of the body of the first if statement since it is evaluated to true. The third version, Path c, contains the statement of the body of the second if statement. Finally, the fourth version, Path d, contains the statements of both if statements. Note, the while loop is contained in all versions since its conditional expression is time related, *i.e.*, not empty.

3.3 SMT Translation and Verification

The last step of our approach generates the constraints for the SMT solver for each path of the program created in the previous step, and incrementally verifies them as depicted by Algorithm 1. For the verification, we use Z3 [4, 10] with its particular extension called Z3opt [3], a state-of-the-art SMT solver that extends the Z3 language to solve linear integer problems.

The translation of a program into a Z3opt model is performed according to the following set of rules:

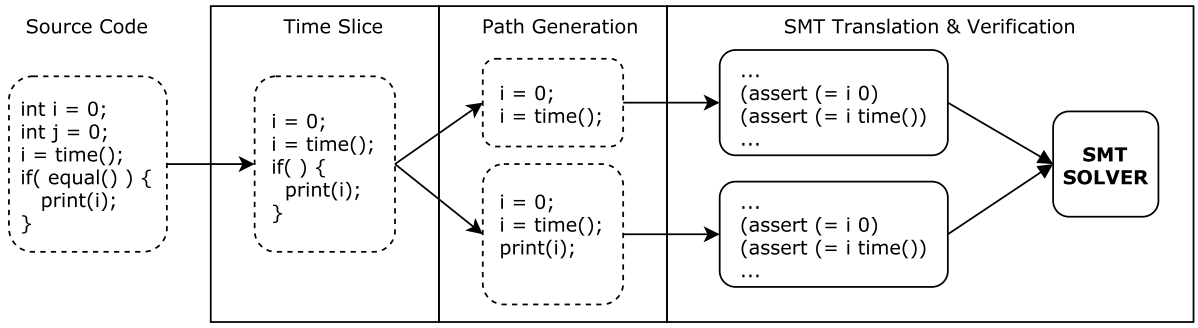

Figure 1: Overview of our approach for detecting time property violations. Time Slice slices the program keeping only the statements that alter and use time variables. Path Generation generates multiple execution paths of the slice based on its control flow. SMT Translation translates each path into SMT constraints. Each model of SMT constraints is fed into an SMT solver to check for violations of time properties in the source code.

```
1  now = time.milliseconds();
2  deadline = now + timeout;
3  while (now <= deadline) {
4    if () {
5      now = time.milliseconds();
6    }
7    if () {
8      now = time.milliseconds();
9    }
10   pollHeartbeat(now);
11   remaining = Math.max(0, deadline - now);
12   client.poll(Math.min(remaining,
         timeToNextHeartbeat(now)));
13   now = time.milliseconds();
14 }
```

Listing 2: Time slice of the source code of the method `poll()` of the issue KAFKA-4290. Statements that do not use or alter time variables are filtered.

Time Variables. For each time variable encountered in the program, our algorithm creates a corresponding SMT variable and two constraints to bound its possible values to be inside the integer domain of Java allowing overflows (lines 5–7 in Algorithm 1). An example of this translation for the time variable `timeout` is presented in Listing 3 in the lines 9–11.

Assignment Statements. Each assignment statement is translated into an SMT assertion that fixes the left-hand-side variable to be equal to the expression of its right-hand-side (line 9 in Algorithm 1). Line 17 in Listing 3 shows the result of translating the assignment statement `deadline = now + timeout`.

Conditional Statements/While Loops. Each condition remaining in a program can be a guard in an if- or while-statement that references time variables. They represent time constraints and therefore can be directly translated into an SMT assertion expressing the condition. The translation of conditions is also handled by line 9 in Algorithm 1. In addition, whenever the condition belongs to a while-loop statement, we obtain the list of time variables referenced

Algorithm 1: Incrementally translate a program P into an SMT model and verify its correctness

```
1  VerifyProgram (P)
       inputs : A program P to verify
       output : A report to identify the error
2      V^t ← getTimeVariables(P);
3      S^t ← getStatements(P);
4      C ← ∅;
5      foreach v_t ∈ V^t do
6          buildConstraint(C, v_t);
7      end
8      foreach s_t ∈ S^t do
9          buildConstraint(C, s_t);
10         if isMethodCall(s_t) then
11             expr_t ← getTimeExpression(s_t);
12             foreach e_t ∈ expr_t do
13                 verify(C, e_t);
14             end
15         end
16         if isLoop(s_t) then
17             var_t ← getTimeGuardVars(s_t);
18             foreach v_t ∈ var_t do
19                 verify(C, v_t);
20             end
21         end
22     end
23 verify (C, expr)
       inputs : An expression expr to maximize and minimize
                with the set of constraints C
       output : A report to identify the error
24     min ← minimize(C, expr);
25     max ← maximize(C, expr);
26     if min < 0 ∨ max > MAX_VAL then
27         reportError(expr);
28     end
```

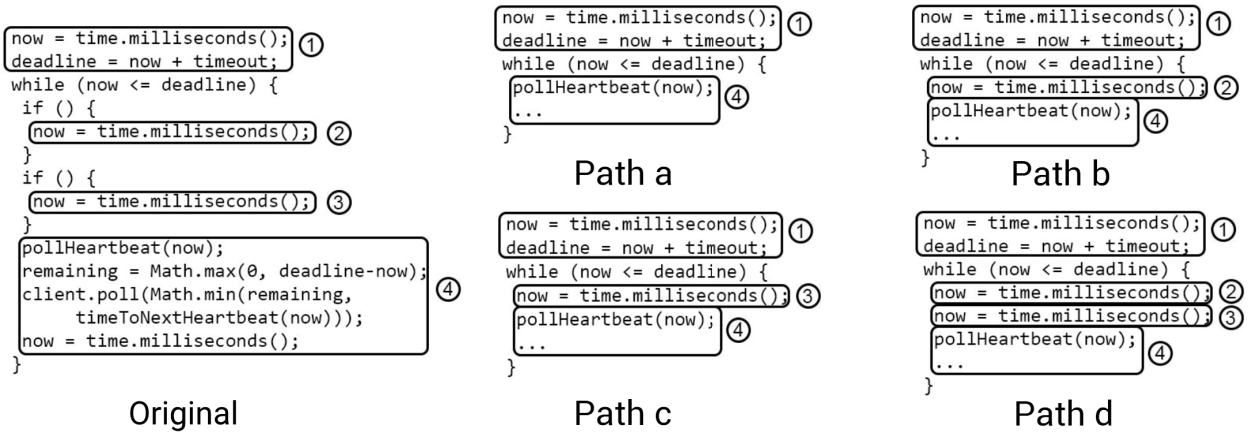

Figure 2: The four different execution Paths a, b, c, and d of the method `poll` created during the Path Generation step. Each path considers a different linearization of the method depending on the execution path taken. The code on the left represents the time slice from Listing 2.

```
1  (declare-const max_val () Int)
2  (declare-const over_max_val () Int)
3  (declare-const min_val () Int)
4  (declare-fun milliseconds () Int)
5  (assert (= max_val 9223372036854775807))
6  (assert (= over_max_val 9223372036854775808))
7  (assert (= min_val (- 9223372036854775808)))
8  (assert (and (>= milliseconds 0) (<= milliseconds max_val)))
9  (declare-const timeout Int)
10 (assert (<= min_val timeout))
11 (assert (>= over_max_val timeout))
12 (declare-const now Int)
13 (assert (= now milliseconds))
14 (assert (<= now over_max_val))
15 (assert (>= now min_val))
16 (declare-const deadline Int)
17 (assert (= deadline (+ now timeout)))
18 (assert (<= deadline over_max_val))
19 (assert (>= deadline min_val))
20 (push)
21 (maximize now)
22 (check-sat)
23 (pop)
24 (push)
25 (minimize now)
26 (check-sat)
27 (pop)
28 (push)
29 (maximize deadline)
30 (check-sat)
31 (pop)
32 (push)
33 (minimize deadline)
34 (check-sat)
35 (pop)
36 ...
```

Listing 3: Excerpt of the SMT Model generated from the `poll()` method as presented in Path a of Figure 2.

in the condition (line 17 in Algorithm 1) and for each one, the verify function is called (lines 18–20 in Algorithm 1).

Method Calls. The translation of method calls, of which at least one argument represents a time expression, is performed in lines 9–15 of Algorithm 1. The algorithm collects each time-related argument (line 11 in Algorithm 1) and for each one, it executes the *verify* function to check whether the method is called with valid time values.

SMT Expressions. The `buildConstraint` function in line 9 of Algorithm 1 is responsible for translating Java statements into SMT constraints. Other than the previous statements, this function has to handle also the different Java expressions that manipulate the time variables. Those expressions denote method calls or mathematical expressions that return or alter time. Mathematical expressions and functions supported by Java are translated one-to-one into the corresponding function provided by the Z3 language. Furthermore, each method call that returns time is translated into a corresponding function in Z3. For instance, the call to the return time category method `time.milliseconds()` in line 1 of Listing 2 is translated into the `milliseconds()` Z3 function in line 4 of Listing 3. For each Z3 function created, our algorithm adds SMT assertion statements to bound its result to be a valid time value, *i.e.*, a value between 0 and the maximum value of a positive integer in the Java language (`max_val` defined in line 5 of Listing 3), modeling the behavior exported by such methods. Line 13 in Listing 3 shows the result of translating the expression in line 2 of Listing 2, in which the return value of the call to method `time.milliseconds()` is assigned to the variable `now`.

Verify Function. Lines 23–29 of Algorithm 1 present the verify function that checks the existence of errors in the constructed

model. The function is called only when time variables are used in a while loop condition or a method call is performed with some time related arguments. In both cases, the function verifies that the time variable or the expression cannot overflow or be negative. For this, the algorithm commands the SMT solver to maximize and minimize the specific time expression given the current set of translated constraints (see lines 24 and 25 in Algorithm 1). If the SMT solver shows that the expression reaches an overflow value or becomes negative (line 26), it records that there is at least one case in which the program can enter in a state where the time semantic cannot be held. If such a violation is detected, the algorithm reports the class, method, time variable, and line of the method call or while loop in which the error occurs.

Listing 3 shows an excerpt of the SMT model created by our approach for the first couple of lines of code of Listing 2. Using the rules above on the first two code lines, our algorithm outputs the lines 1–19 in Listing 3. Next, our algorithm processes the while loop in line 3 of Listing 2. Since it is a while loop, our algorithm calls the `verify` function for each variable referenced in its guard expression. Lines 20–27 in Listing 3 show the output for the `now` variable. Since it stores the timestamp returned by the call to `time.milliseconds()`, the SMT solver verifies it to be correct. On the contrary, in line 29 of Listing 3 the maximization of the time variable `deadline` detects an overflow error caused by the sum of `now` and `timeout`. Our algorithm stops here and reports, that method `poll()` in class `WorkerCoordinator` contains an overflow error which makes variable `deadline` in line 4 to store an invalid time value.

4 EXPERIMENTS

In this section, we present the experiments we have performed to evaluate our approach. We use two empirical experiments to measure the precision and runtime performance of our implemented approach using 20 Java open source projects. In summary, our evaluation aims to answer the following two research questions:

- ***RQ1***: What is the precision of our approach in detecting time related errors in source code?
- ***RQ2***: What is the run time that our approach requires for producing the results?

4.1 Experimental Setup

We design an empirical experiment in RQ1 and RQ2 with 20 open source Java projects. We selected Java projects that use multi-threading and distributed components to maximize the presence of statements that deal with time. The selected projects also differ in vendor, size, domain of use, and coding convention adopted to maintain and develop them. We also considered projects that are stand-alone applications and projects that are frameworks used to develop applications. Using these criteria resulted in the set of 20 Java projects listed in Table 1. In sum, they comprise 90,908 source files implementing 125,130 classes containing 939,861 methods and more than 9.5M SLOCs. We conducted the experiments on a computer with a 2.5GHz Intel CPU with 16GB of physical memory running macOs 10.12.6.

Table 1: List of Java projects used for the evaluation together with their number of files, number of classes, number of methods, and Source Lines of Code (SLOC).

Name	Files	Classes	Methods	SLOC
activemq	4,434	4,981	41,212	415,976
Activiti	2,002	2,103	15,358	139,672
airavata	1,621	9,320	70,843	711,587
alluxio	1,319	3,364	24,859	233,897
atmosphere	348	500	4,043	35,843
aws-sdk-java	26,416	27,208	205,202	1,795,234
beam	1,696	3,844	20,477	210,960
camel	17,205	20,024	114,938	1,065,292
elastic-job	571	611	2,493	26,418
flume	642	995	6,627	85,750
hadoop	8,063	12,605	99,343	1,271,230
hazelcast	5,696	7,663	58,405	649,789
hbase	3,638	9,535	127,061	1,201,149
jetty.project	2,567	3,781	24,907	342,602
kafka	1,315	1,896	13,669	149,644
lens	845	1,036	8,063	99,523
nanohttpd	87	124	710	7,532
neo4j	6,681	9,158	60,378	680,986
sling	5,336	5,964	36,969	427,779
twitter4j	426	418	4,304	32,436
Overall	90,908	125,130	939,861	9,583,299

4.2 RQ1: What is the precision of our approach in detecting time related errors in source code?

The first experiment aims to evaluate the precision of our approach in detecting the usages of time variables in statements that can store invalid time values. For this, we ran our prototype tool on the source code of each Java project and collected the reports of all detected errors. Next, we manually analyzed all the reports and verified in the source code the reasons that led to the reported error. For each reported time variable, we manually analyzed its dataflow and we verified if there is a possibility that the variable could store an invalid time value when it is used. We computed the precision by counting the number of correct errors manually found over the total number of errors reported by our approach. Note, we did not verify if the reported error results in a true failure of the program, because we do not have the full set of specifications. We mark an error as true positive if there is a case in which the variable could store an invalid time value, while false positives are errors reported by our tool which are not real errors because developers handle them in a different section of the program. For each verified report, we open an issue on the respective project's issue tracker.

Table 2 presents the results of our evaluation. Over all projects, the prototype tool analyzed 939,861 distinct Java methods, out of which 466,218 contain at least one statement that deals with time. On average, almost half (49.6%) of the methods deal with time resulting in a total of 690,008 paths to analyze. In two projects,

Table 2: Results of the error detection for the 20 Java projects showing number of analyzed methods (#Methods), number of methods dealing with time (#T. Methods), number of paths created from time slices (#Paths), number of SLOC for the paths generated, percentage of SLOC generated compared to the project SLOC (% SLOC), number of detected time related errors (#Detected), number of real errors (#TP), number of false positive errors (#FP), and the precision computed by #TP/(#TP+#FP).

Name	#Methods	#T. Methods	#Paths	SLOC	% SLOC	#Detected	#TP	#FP	Precision
activemq	41,212	12,583 (30.5%)	16,447	38,430	9.24%	16	13	3	81.250%
Activiti	15,358	6,034 (39.3%)	7,885	15,798	11.31%	0	0	0	-
airavata	70,843	39,858 (56.3%)	70,015	646,626	90.87%	0	0	0	-
alluxio	24,859	13,570 (54.6%)	19,706	84,268	36.03%	0	0	0	-
atmosphere	4,043	1,626 (40.2%)	2,237	3,875	10.81%	1	1	0	100.000%
aws-sdk-java	205,202	150,932 (73.6%)	247,855	2,227,270	124.07%	1	1	0	100.000%
beam	20,477	7,832 (38.2%)	9,489	8,125	3.85%	0	0	0	-
camel	114,938	34,760 (30.2%)	44,960	87,985	8.26%	4	4	0	100.000%
elastic-job	2,493	637 (25.6%)	783	559	2.12%	0	0	0	-
flume	6,627	2,429 (36.7%)	3,614	9,771	11.39%	3	3	0	100.000%
hadoop	99,343	40,173 (40.4%)	54,819	121,523	9.56%	27	26	1	96.296%
hazelcast	58,405	20,741 (35.5%)	25,488	36,626	5.64%	17	14	3	82.353%
hbase	127,061	81,747 (64.3%)	111,069	175,268	145.92%	24	22	2	91.667%
jetty.project	24,907	8,057 (32.3%)	12,779	37,794	11.03%	14	13	1	92.857%
kafka	13,669	5,158 (37.7%)	7,196	13,616	9.10%	13	13	0	100.000%
lens	8,063	3,917 (48.6%)	5,265	10,450	10.50%	0	0	0	-
nanohttpd	710	205 (28.9%)	294	659	8.75%	0	0	0	-
neo4j	60,378	18,595 (30.8%)	24,435	35,425	5.20%	5	4	1	80.000%
sling	36,969	15,489 (41.9%)	23,111	57,342	13.40%	21	20	1	95.238%
twitter4j	4,304	1,875 (43.6%)	2,561	12,663	39.01%	0	0	0	-
Overall	939,861	466,218 (49.6%)	690,008	3,624,073	28.30%	146	134	12	**91.781%**

`aws-sdk-java` (127.07%) and `hbase` (145.92%), the Path Generation step creates more lines of code to analyze compared to the original project size. On all the other projects, except `airavata` (90.87%), `alluxio` (36.03%), and `twitter4j` (39.01%), the number of lines of code to analyze is smaller than 13.40% of the original project size.

In this set of methods, our approach discovered 146 time related errors. The three projects with most of the errors are `hadoop` (27), `hbase` (24), and `sling` (21). In 8 out of the 20 projects, our approach could not detect any errors. In two projects, namely `atmonsphere` and `aws-sdk-java`, our approach detected only one error.

We manually analyzed the source code of each error and confirmed 134 of the 146 as real errors. For instance for `hadoop`, we confirmed 26 out of the 27 detected time related errors. Overall, only 12 errors reported by our approach were found to be false positives. This results in a precision of 91.781% on average. For only three projects, namely: `activemq`, `hazelcast`, and `neo4j` we obtained a precision below 91%.

Through the manual analysis of the errors in the source code, we discovered cases in which developers added a comment to the code stating that they know that the time properties are not preserved and they justified why they think that it is not a problem. An example of such a justification is that they check the correctness of the time variable later on in the program. In cases where the developers did not provide an explanation in the source code, we filed a corresponding bug report in the issue tracker of the project. For one such bug report, a developer answered with the comment: *"Setting a negative* `initialDelayTime` *is an error so an exception should be thrown indicating this so the value can be fixed"*. While this response shows that the developer is aware of the potential error, it also confirms that our approach is capable of detecting these errors.

4.3 RQ2: What is the run time that our approach requires for producing the results?

In addition to the low rate of false positives, it is also important for the usability of our approach in practice that it returns the results within a reasonable amount of time. We envision our approach to be integrated into build environments so that the detection of errors can be performed in the build and testing stage, or, if performance allows, even within development environments. In the latter case, the response time of our approach should be within 60 seconds, and ideally even within 10 seconds [23].

In our approach, we use the SMT solver Z3 as oracle to formally verify the correctness of time properties in Java methods. It is well known that SMT solvers can consume a lot of resources and time to perform the verification since a huge state space needs to be explored and usually a timeout is enforced. In our experiments, all the models that are created do not specify any timeout. In addition, our approach performs the verification of multiple copies of the source code generated from a time slice of the program. This means on the one hand, verifications are performed with a reduced state space but on the other hand multiple verification runs need to be performed to cover all possible paths of the program execution.

Using the data from the 20 Java open source projects, we evaluated the response time of our approach. For each project, we

Table 3: Runtime of our approach in seconds for parsing the 939,861 methods of the 20 Java projects and detecting errors.

	Time spent in seconds (s)			
Name	Total	Parsing	Detecting	ms/method
activemq	243.8	226.5	17.3	5.92
Activiti	137.6	131.4	6.2	8.96
airavata	505.6	460.5	45.1	7.14
alluxio	149.4	130.1	19.2	6.01
atmosphere	24.5	22.3	2.3	6.06
aws-sdk-java	4,515.4	4373.5	141.9	22.00
beam	131.5	125.4	6.1	6.42
camel	1,361.2	1333.1	28.2	4.84
elastic-job	11.5	10.8	0.7	4.62
flume	22.8	20.1	2.7	3.44
hadoop	598.7	542.3	56.4	6.03
hazelcast	281.3	266.3	15.0	4.82
hbase	1,014.0	923.6	90.4	7.98
jetty.project	168.9	158.7	10.2	6.78
kafka	62.1	55.0	7.1	4.55
lens	36.6	32.9	3.7	4.54
nanohttpd	3.7	3.1	0.6	5.25
neo4j	264.4	247.6	16.8	4.38
sling	282.3	268.1	14.2	7.64
twitter4j	26.1	24.3	1.8	6.07
Overall	9,841.4	9,355.6	485.8	(AVG) **7.02**

measured (i) the time required to parse the source code, and (ii) the time to detect methods that deal with time and analyze them with respect to the correct usage of timestamps (*i.e.*, the defined semantics of time in Java 8).

The time for parsing the source code comprises the time that our approach needs to construct the abstract syntax tree (AST) using the Eclipse JDT library. The time for detecting the errors comprises the various steps of our approach to create the time slice, the creation of the paths, the translation of each path into an Z3opt model, the incremental verification of these models, and the report of the results. Table 3 summarizes the results of running our approach on the 20 projects. The results show that the experiment in total required 9, 841.4 seconds (~2.73 hours) to process all the 939,861 Java methods. Overall, the parsing required 9, 355.6 seconds while the detection only took 485.8 seconds. As expected, the largest projects, namely `aws-sdk-java` (4,373.5) and `camel` (1,333.1), took the longest to parse. A large extent of the time for the parsing is dedicated to the resolution of the binding information of the method calls in Eclipse JDT.

The performance of the steps to detect errors of time properties is remarkably fast. For instance, it took our approach 141.9 seconds to detect errors in 205,202 methods of the `aws-sdk-java` project which is on average 22.00 ms per method. Over all projects, the detection takes on average 7.02 ms per method. One reason is that time slices are only created for methods that contain time statements which are 49.6% of the methods (see Table 2). Another reason is that, even though we generate multiple copies of the time slice, the total number of SLOC to analyze is just 28% of the total SLOC of the projects source code.

5 DISCUSSION & THREATS TO VALIDITY

This section discusses the results of our study and limitations of our current approach. Furthermore, we discuss potential threats to validity in our experiments.

5.1 Discussion

In this section, first we present the theoretical foundation of our technique and second, we discuss the results obtained with our experiments.

From a theoretical point of view, our approach is sound but not complete. The theoretical foundation of our approach is rooted in the formal time semantics of Java 8. The soundness of our approach assures that every detected error is indeed an error w.r.t. our definition of the formal semantics of timestamps as integer values in Java. The soundness proof is essentially a structural induction proof on the structure of time related Java statements based on the operational semantics of Java [5] and formal semantics of time [18]. In fact, the time slice extracted by our approach from a Java method preserves the time semantics of the original source code. Moreover, the aforementioned semantics are preserved by the translation of the time slice into the SMT model. However, the further details of the proof are beyond the scope of this paper.

The completeness of our approach cannot be established mainly because of the following two reasons. First, our approach may extract a weak model since we do not consider the full specification of the program. Second, for extracting the SMT models our approach only considers calls to methods of the category Return Time, while still the method calls belonging to other categories may alter the value of a class attribute that represents time. Therefore, we cannot assure the absence of false positives.

To complement the theoretical foundation of our approach, we have investigated the precision (computed with the standard formula of true positives over the sum of true positives and false positives) of our technique with an empirical study. With our evaluation we show that we could not reach a perfect precision which confirms that our approach is not complete. The small amount of false positives that we found, as we described in Section 4.2, are due to the light-weight dataflow analysis performed by our approach. Notwithstanding, we achieved roughly 91.78% of correct detection of time related errors with only 12 false positives, therefore we can provide developers with an automatic tool for effectively identifying time-related errors in Java programs. Furthermore, at the moment of writing, developers indicated the usefulness of our tool. They confirmed the existence of an error that was discovered with our approach. The other bug reports are still pending to be verified by the developers of the projects. We did not find errors in every project because many of those have a long history of development and therefore, the critical sections, such as time related functionalities, are well implemented and mature.

In addition, we want to provide an approach that can be integrated into a development environment. Therefore, the approach should be scalable and produce the results in a reasonable amount of time. The construction of the slices of a Java program with only

time related statements paid off in performance. In fact, our approach needs to process on average only 28.30% of the source code. This enables our approach to process a large amount of methods per second. The prototype tool spent 95% of the time in parsing the source code and constructing the abstract syntax tree with the Eclipse JDT library. Only 5% of the time was used to construct the model of the code and to verify the time constraints with the SMT solver. The construction of the AST is necessary to every static analysis technique and we cannot skip this part. Without a processable representation of the source code we cannot provide any kind of analysis of it. Despite this overhead, our approach detects errors with high degree of rigor requiring on average only 7.02 ms to process a method, which other automated error detecting approaches, such as Randoop, cannot yet achieve, even given indefinite time.

In summary, we showed that our approach can effectively discover time errors transforming the time semantics of Java programs into a set of SMT constraints. Furthermore, our approach can be integrated into other state of the art tools for error discovery, such as Randoop [19], Agitator [6], or SMACK [7].

5.2 Threats To Validity

In the following section, we discuss threats to the internal and external validity of our evaluation and how we addressed them in our experiments.

Internal Validity. The internal validity threat indicates the reliability of our prototype implementation and experiments.

One limitation of our approach is the approximated model of the program, *i.e.*, it may not correctly handle loops because it translate the execution for a single iteration only. Our experiment with the 20 Java open source projects shows that this approximation works well since we managed to discover multiple errors in loops. This limitation can be removed by identifying the invariants that control the number of loop iterations. The identification of invariants with static analysis is, however, a hard problem. Therefore, we made a trade-off between time/space complexity and completeness considering a single iteration only. Future works will be devoted to address this problem, employing dynamic analysis to discover likely invariants or asking developers to provide them.

Furthermore, our approach currently supports only time APIs of the Java 8 standard library. We do not consider other Java libraries providing time APIs, such as Joda-Time.[3] However this threat is mitigated by our findings, since in our study with the 20 Java projects, we discovered that none of them uses external libraries for handling time. In addition, with JSR310,[4] Java 8 added better date and time APIs, and most likely, Java developers will stop using external libraries for implementing time behavior in their programs.

In the second research question, we studied the impact of the static analysis approach on the runtime of our prototype tool. We found that in particular the parsing of the source code required 95% of the time which might have been due to using the parsing library Eclipse JDT. In future work, we plan to address this threat by using also other parsing libraries for implementing our approach, such as Java Parser.[5]

[3] http://www.joda.org/joda-time/
[4] https://jcp.org/en/jsr/detail?id=310
[5] https://javaparser.org/

External Validity. The external validity threat concerns the generalization of the results to other software projects in two dimensions: (i) the effect of application of our methodology to new datasets and (ii) the extendibility of the approach to other languages.

The results of our evaluation can be easily generalized to other Java projects because our approach is sound with the respect to the time semantics of the Java programming language. However, our presented technique is not complete and we mitigated this threat investigating how much this affects the precision of our approach. We applied the implementation of approach to 20 open source Java projects that differ in vendor, size, domain of use, and coding convention adopted to maintain and develop the system. Moreover, we considered projects that are stand-alone applications and also frameworks used to develop other applications.

The implementation of our approach in a prototype tool can be further applied to other Java projects to extend our study and validate our findings. Furthermore, our approach can be adapted to other programming languages, such as C#, that use similar mechanisms to implement timestamps as used by the Java programming language. This would mainly require the adaptation of our definition of semantics for time and the change of our parsing library to support parsing of source code written in other languages.

6 RELATED WORK

A wide spectrum of related work in literature addresses automated error detection in programs. They can be divided into three different approaches: static analysis, testing, and verification.

Static Analysis. Several static analysis tools for various programming languages have been developed. Basically, they analyze the source code of a project and apply specific syntactical rules to detect problems that could lead to errors. For instance, FindBugs [17] and PMD [9] are two well know examples that analyze the source code to find common programming flaws, such as unused variables, empty catch blocks, and unnecessary object creation. Similarly, JLint [1] analyzes Java code to detect synchronization problems.

Testing. Several approaches to automate the generation of (unit) test cases for (object-oriented) programming languages have been investigated. Instead of looking for specific patterns in the source code, those techniques try to identify unexpected behavior of a given method. For instance, Randoop [19–22] is a well known tool that, given a program, can be used to find bugs in it and to create regression tests to warn developers about erroneous changes of the program behavior. Randoop generates unit tests using feedback-directed random test generation to create sequences of method/constructor invocations for the classes under test. Havrikov *et al.* [14] consider the specific domain of XML Schema definition for which they can create a set of unit tests to achieve high test coverage. Other commercial tools, such as Agitator [6], first, use dynamic analysis to discover invariants in the program and then, they create unit tests that assert those invariants reaching a coverage of around 80%. However, all these approaches cannot detect time related errors because they do not consider the specific semantics of the language and they lack of a sound verification background. In fact, although test coverage is a good metric for the likelihood to

have tested the code in different scenarios, it does not provide any guarantee that a program has no errors.

Verification. Many approaches exist that verify source code to detect errors in programs, however, only few of them address time properties. Most of the existing work is on discovering race conditions and synchronization problems, looking at the timing on which different memory accesses occur during the execution of a program. Java Pathfinder [13] is a tool developed by NASA. It executes the bytecode of a given program in a special virtual machine that is capable of verifying properties of the bytecode with a focus on race conditions. Similarly, Bandera [12] extracts a formal representation from Java bytecode that is converted into the SPIN model checker [16] to verify that the time sequence of actions in the program execution respects the given specification. Walkinshaw *et al.* [24] present an extension of state machine inference from a program execution that accounts for temporal properties of the system. The work of Henzinger *et al.* [15] provides a technique to verify temporal events, such as the correct execution order of the mutex API. All these existing approaches have in common that they address time as ordered sequence of events that occur in the program execution. In contrast, in our work, we address time as domain that can be altered by statements in the program and not as sequence of events.

7 CONCLUSION

In this paper, we presented an approach to detect time related errors in Java programs. We showed the problem through a bug taken from an open-source Apache project and we also showed how existing tools fail to detect it, mainly because they do not consider the semantics of time.

We presented an approach that automatically identifies time related errors in Java methods. Our approach uses the formal time semantics of Java version 8 to identify time related statements and variables in a Java method. These statements and variables are translated into a set of SMT constraints that then are formally verified by an SMT solver which detects and reports errors according to the given time semantics.

We performed two experiments to evaluate the precision and runtime performances of our approach on 20 open source Java projects. Our results show a low rate of false positives and appropriate scalability. Our approach benefits developers with an automatic verification technique that helps them to identify time related problems. With our study, we show that it is able to correctly identify time related errors with a precision of 91.781% with only 12 false positives. Moreover, the implementation of our approach and data used in our experiments are publicly available.[1]

Future work will be performed in two directions. First, we plan to add dynamic analysis to our approach to discover likely invariants. This way, we can better model loop statements. We also plan to improve our inter-procedural data flow analysis to reduce the number of false positives detected by our approach. Second, we plan to integrate our approach into a development environment and extend our study to other programming languages that use a time semantics similar to Java.

ACKNOWLEDGMENT

This research is funded by the Austrian Research Promotion Agency FFG within the FFG Bridge 1 program, grant no. 850757.

REFERENCES

[1] C Artho. 2006. JLint – Find Bugs in Java Programs. (2006). http://jlint.sourceforge.net/.
[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.
[3] Nikolaj Bjørner and Anh-Dung Phan. 2014. νZ-Maximal Satisfaction with Z3.. In *In Proceedings of the 6th International Symposium on Symbolic Computation in Software Science (SCSS)*, Vol. 30. 1–9.
[4] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ-An Optimizing SMT Solver.. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Vol. 15. Springer, 194–199.
[5] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 445–456.
[6] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. 2006. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA)*. ACM, 169–180.
[7] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamarić, and Michael Emmi. 2016. SMACK software verification toolchain. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 589–592.
[8] Robert N Charette. 2005. Why software fails. *IEEE spectrum* 42, 9 (2005), 36.
[9] Tom Copeland. 2005. *PMD Applied*. Centennial Books.
[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.
[11] David Deharbe, Pascal Fontaine, and Bruno Woltzenlogel Paleo. 2011. Quantifier inference rules for SMT proofs. In *First International Workshop on Proof eXchange for Theorem Proving (PxTP)*.
[12] John Hatcliff and Matthew Dwyer. 2001. Using the Bandera tool set to model-check properties of concurrent Java software. In *International Conference on Concurrency Theory (CONCUR)*. Springer, 39–58.
[13] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)* 2, 4 (2000), 366–381.
[14] Nikolas Havrikov, Alessio Gambi, Andreas Zeller, Andrea Arcuri, and Juan Pablo Galeotti. 2017. Generating unit tests with structured system interactions. In *Proceedings of the 12th International Workshop on Automation of Software Testing (AST)*. IEEE Press, 30–33.
[15] Thomas A Henzinger, George C Necula, Ranjit Jhala, Gregoire Sutre, Rupak Majumdar, and Westley Weimer. 2002. Temporal-safety proofs for systems code. In *International Conference on Computer Aided Verification (CAV)*. Springer, 526–538.
[16] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.
[17] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM Sigplan Notices* 39, 12 (2004), 92–106.
[18] Giovanni Liva, Muhammad Taimoor Khan, and Martin Pinzger. 2017. Extracting timed automata from Java methods. In *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 91–100.
[19] Carlos Pacheco and Michael D Ernst. 2005. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 504–527.
[20] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA)*. ACM, 815–816.
[21] Carlos Pacheco, Shuvendu K Lahiri, and Thomas Ball. 2008. Finding errors in. net with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA)*. ACM, 87–96.
[22] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*. IEEE Computer Society, 75–84.
[23] Steven C. Seow. 2008. *Designing and Engineering Time: The Psychology of Time Perception in Software* (1 ed.). Addison-Wesley Professional.
[24] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring finite-state models with temporal constraints. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 248–257.