Automatic Repair of Timestamp Comparisons

Giovanni Liva, *Member, IEEE,* Muhammad Taimoor Khan, *Member, IEEE,*
Martin Pinzger, *Senior Member, IEEE,* Francesco Spegni, and Luca Spalazzi

Abstract—Automated program repair has the potential to reduce the developers' effort to fix errors in their code. In particular, modern programming languages, such as Java, C, and C#, represent time as integer variables that suffer from integer overflow, introducing subtle errors that are hard to discover and repair. Recent researches on automated program repair rely on test cases to discover failures to correct, making them suitable only for regression errors. We propose a new strategy to automatically repair programs that suffer from timestamp overflows that are manifested in comparison expressions. It unifies the benefits of static analysis and automatic program repair avoiding dependency on testing to identify and correct defected code. Our approach performs an abstract analysis over the time domain of a program using a Time Type System to identify the problematic comparison expressions. The repairing strategy rewrites the timestamp comparisons exploiting the binary representation of machine numbers to correct the code. We have validated the applicability of our approach with 20 open source Java projects. The results show that it is able to correctly repair all 246 identified errors. To further validate the reliability of our approach, we have proved the soundness of both, type system and repairing strategy. Furthermore, several patches for three open source projects have been acknowledged and accepted by their developers.

Index Terms—Software/Program Verification, Formal methods, Error handling and recovery

1 Introduction

Software failures are expensive and they consume the majority of developers time [1], nonetheless most of these software failures are predictable and avoidable [2]. Researchers have proposed several analysis techniques of source code to reduce software maintenance and fixing effort to remove implementation errors.

A technique to reduce the developers' effort to fix a defected implementation is automated program repair. Briefly, an automated program repair approach performs some transformation of the source code to remove an existing error. This technique is performed in two steps: first it identifies statements in the source code that contain errors and then, it repairs them. The modern repair techniques [3], [4], [5], [6], [7], [8] often use test cases to (i) construct models of the correct behavior of a program to identify the errors to repair; and to (ii) validate their proposed repairs. Thus, such techniques require developers to write reproducible and deterministic tests that point out the error subjected to the repair. This helps to solve issues due to regression errors but it is not feasible for new yet-undiscovered faults. Other approaches, such as Randoop [9], [10], Agitator [11], or Evosuite [12] help developers to automatically create test cases for discovering new errors. The generated tests stress functionalities of the program with random sequences of input values and method invocations for the class under test. They help developers achieve a high coverage [13] and, hopefully, discover previously unknown defects that can be addressed by the automated program repair techniques.

- *G. Liva, and M. Pinzger are with the Department of Software Engineering, Alpen-Adria Universität Klagenfurt, Austria.*
  E-mail: {Giovanni.Liva, Muhammad.Khan, Martin.Pinzger}@aau.at
- *M. T. Khan is with the School of Computing and Mathematical Sciences, University of Greenwich, London, UK.*
  E-mail: m.khan@gre.ac.uk
- *F. Spegni, and L. Spalazzi are with the Department of Information Engineering, Università Politecnica delle Marche, Italy.*
  E-mail: {f.spegni,l.spalazzi}@univpm.it

It is well known that relying on testing for the identification of errors has a shortcoming: they show the existence of errors but they do not provide insight about what is their root cause. Therefore, testing activities have been complemented with static analysis [14] that considers the semantics of the code. This type of technique performs symbolic execution or an abstract interpretation of the program aiming at finding errors as early as possible before shipping the program to the customers. Huge software companies, such as Google [15], [16], Facebook [17], [18], and Microsoft [14], [19] are pushing the integration of static analysis tools in their development cycle. With the popularity of continuous integration and continuous delivery systems (CI/CD), it is important to automatically prevent faults and these tools fit perfectly in this software development practice. For every release of the software, a verification pipeline is executed where the provided tests and static analysis tools are executed. If at any point errors are found, the building process is terminated and the executable is not delivered to the users. Moreover, the results of the verification pipeline are sent to developers to help them fix the identified bugs.

Identifying and repairing errors in a program is challenging, even more when the error stems from bad handling of timing of events. Modern programming languages, such as Java, C, and C#, offer APIs to manipulate and model time as timestamp using integer variables. Recent works [20], [21], [22] show how the timestamp representation is fragile in the context of mainstream programming languages. Integer Overflows due to manipulation of timestamps could be dangerous and exploited to violate the security offered by modern operating systems. Recently, two vulnerabilities[1,2] due to timestamp overflows were discovered in the Linux kernel. An example of the problems inherited by manipulating time via timestamp is presented in Listing 1 that

1. https://nvd.nist.gov/vuln/detail/CVE-2018-12896
2. https://nvd.nist.gov/vuln/detail/CVE-2018-13053

```
1  public T acquire(long time, TimeUnit unit)
       throws TimeoutException, IOException {
2    long endTimeMs = mClock.millis() + unit.
         toMillis(time);
3    ...
4    long currTimeMs = mClock.millis();
5    try {
6      if (currTimeMs >= endTimeMs || !mNotEmpty.
           await(endTimeMs - currTimeMs, TimeUnit.
           MILLISECONDS)) {
7        throw new TimeoutException("Acquire
             resource times out.");
8      }
9    }
10   ...
11 }
```

Listing 1: Excerpt of method `acquire` that contains a timestamp overflow error.

shows a bug of Alluxio,[3] discovered and repaired with our approach. The method `acquire` accepts a timeout parameter that expresses the maximal amount of time that the caller is willing to wait for acquiring a resource. The method implements the acquisition with a while *true* loop (omitted) that iterates until either the resource is acquired or it times out throwing an exception. Variable `endTimeMs` (see line 2) contains the expiration date that is used to verify whether the request times out. It is computed as the sum of current time and the `timeout` parameter. Since timestamp stores the milliseconds that have been passed since January 1st 1970, those values are inherently huge and can easily overflow. In fact in the timestamp comparison at line 6, if the variable `endTimeMs` previously overflowed in line 2, the method wrongly returns the timeout exception without waiting for the resource to be available for the expected amount of time.

We propose a novel approach to automatically repair programs that suffer from timestamp overflows errors that are manifested in comparison expressions. It combines the benefits of static analysis and automatic program repair and it can be easily integrated into the CI/CD pipeline of a project, without depending on testing to identify and repair the defected code. Our approach performs an abstract analysis over the time domain of a program using the formal time semantics of a programming language to support a Time Type System (TTS). We show the applicability of our approach to the Java language using the time semantics presented by Liva *et al.* [20]. One peculiarity of the time semantics is that it can be used to identify those program variables that store time values, called time variables. Time variables in Java use the integer representation with either the `int` or `long` data types to store numbers that represent time values. A Java program can handle the time either by looking at the moment when one or more events occur, or by computing the difference between two events. For this reason we refine the time analysis in [23] by recognizing two different types of time variables: **Timestamp** and **Duration**. If we interpret the (real) time as a line, then Timestamp values are used to represent arbitrary points along this line, while Duration values can be used to represent an interval between two points. Timestamp variables are problematic because they store huge numbers and they can easily overflow. Thus, we aim to repair comparison expressions between Timestamp values. We exploit TTS to find fragile Timestamp comparisons in Java programs, such as the example presented in Listing 1, that can be automatically repaired.

We have implemented the approach in an open source prototype tool[4] that we have applied to 20 open source Java projects to study the applicability of our approach. In our evaluation, we answer the following research questions:

- **RQ1:** What is the precision and recall of TTS in inferring time types?
- **RQ2:** What is the correctness of the patches created by our approach?
- **RQ3:** What is the usefulness of the patches created by our approach?

With the first research question, we aim to discover empirically the precision and recall of our time type system to infer the time types of expressions. The second research question is used to assess if the patches can remove the errors and finally, we investigate with the feedback of developers if the patches can be accepted as repair. The results of our evaluation show that TTS has a precision of 100% and a recall of 99.97% in identifying timestamp comparisons. Moreover, all of the proposed patches are correct and several of them have been acknowledged and accepted by the developers of three projects. In summary, this paper makes the following contributions:

- an extension of the formal time semantics for the Java programming language;
- a time type system;
- an evaluation of the approach with 20 open source projects;
- several errors repaired that have been accepted by the developers of three open source projects.

The remainder of the paper is organized as follows: Section 2 presents an overview of our approach; Section 3 details the definition of the time type system and Section 4 presents our repair approach. In Section 5 we evaluate our approach and we discuss implications, limitations, and threats to the validity of our experiments in Section 6. Section 7 gives an overview of the related works and we conclude the paper in Section 8.

2 Approach

Figure 1 presents the two steps of our approach to repair timestamp comparisons. In the first step, presented in Section 3, our approach analyzes the source code of a program to extract the time information. First, it applies the time semantics presented by Liva *et al.* [23] to identify time expressions in the source code. Then, this information is processed by our Time Type System (TTS) that extracts the time type of each expression. These operations are applied recursively until a fix point is reached and no more additional time expressions are identified in the source code. Notice that since the time semantics is complete [23], all time variables and statements will be processed by TTS. The output of this step is the source code annotated with

3. https://github.com/Alluxio/alluxio/pull/7320

4. https://git.io/fNiBz

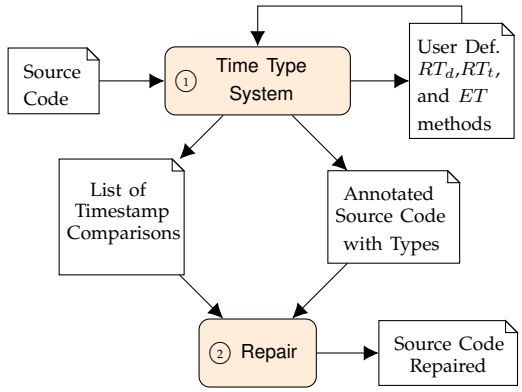

Fig. 1: Overview of our approach to repair timestamp comparisons.

Methods	m	::=	$i(x_1, \_, x_n)\{\ s\ \}$
Statements	s	::=	$e \mid x = e \mid \texttt{if}\ (b)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$
		|	$\texttt{while}\ (b)\ s \mid s_1; s_2 \mid \texttt{return}\ e$
Expressions	e	::=	$n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
		|	$e_1 \div e_2 \mid obj.m(e_1, \_, e_n) \mid b$
		|	$\frac{\texttt{min}}{\texttt{max}}(e_1, e_2)$
Booleans	b	::=	$e_1 < e_2 \mid e_1 <= e_2 \mid e_1 >= e_2$
		|	$e_1 > e_2 \mid b_1\ \texttt{\&\&}\ b_2 \mid b_1\ \texttt{||}\ b_2$

Fig. 2: Programming language grammar supported by the Time Type System (TTS).

the time type information and the list of statements that perform a comparison between two timestamp expressions that we call Timestamp comparisons.

Given the list of comparisons and the source code annotated with time types, the second step repairs all timestamp comparisons that could have overflow problems using the technique described in Section 4.

3 Time Type System

Our approach extends the formal time semantics of the Java programming language proposed by Liva *et al.* [23] to support a Time Type System (TTS). One peculiarity of the time semantics is that it can be used to identify those program variables that store time values, called time variables. In addition to the date/time APIs provided by the Java JDK and other libraries, time variables can be declared using the `int` or `long` data types. Furthermore, such time variables (and expressions) can be used in a program to store two semantically different information:

1) **Duration**: the value of the expression specifies a scalar amount of time;
2) **Timestamp**: the value of the expression specifies a specific point in time.

If we interpret the (real) time as a line, then Timestamp values are used to represent arbitrary points along this line, while Duration values can be used to represent an interval between two points. This distinction is mandatory for our purposes since our repair strategy seeks for expressions that compare Timestamp values that can suffer from integer overflow problems.

In the following subsections, we introduce a type system to infer the two time types for time expressions in programs.

3.1 Time Semantics Extension

The syntax and definition of the time type system abstracts from the definition of a specific programming language. In this manner, it can be used for multiple programming languages providing the mapping between the specific constructs offered by the programming language to our time type system syntax. However, since we show the applicability of our approach for the Java programming language using the time semantics defined by Liva *et al.* [23], we present examples that map Java constructs to the syntax of the time type system. Through a manual analysis of the Java 8 time APIs, the authors have formally defined the semantics of four types of time methods that can be used in Java. On top of this analysis, they have defined rules that infer program variables which store time values, called time variables, and statements that deal with time, called time statements. From [23], we are interested in only the methods that use, or can modify, time variables, namely methods that: (i) return time values (RT) and (ii) accept time as a parameter (ET). We extend their analysis dividing the RT methods into two more fine-grained categories:

- RT_t as the set of RT methods that return a Timestamp value;
- RT_d as the set of RT methods that return a Duration value;

Furthermore, we refine also the set of ET methods of the Java time APIs, for which we manually annotated each time parameter with its time type. For the remaining of the paper, with ET we refer to this extension. An example of our refinement is the method `java.lang.Object.wait(long timeout)` that we categorized as ET method and we assigned the parameter `timeout` the type Duration. As another example, the method `java.lang.System.nanoTime()` is categorized as RT_t since it returns a Timestamp value.

Based on this semantics of manually categorized Java time methods, TTS recursively analyzes the source code of a project until a fix point is reached, *i.e.*, no more time methods are identified. In each iteration, TTS extends the three sets of time methods and when a new time method is found, TTS processes the source code again since the method might be used to discover new time variables, statements, and methods.

3.2 Syntax Rules

Figure 2 presents a subset of the generic programming language supported by TTS described using rules in Backus-Naur form. It presents the most interesting cases of the language with methods, statements, expressions, and boolean comparisons. Here, i represents an identifier for method names and obj for object names, n represents a numerical literal, and x ranges over program variables.

$$\frac{\Gamma_0 := \forall_{i=1}^{n} isTimeVar(x_i) \rightarrow \Gamma[x_i \mapsto DoT] \quad \Gamma_0 \vdash s \dashv \Gamma'}{\Gamma \vdash i(x_1, \_, x_n)\{\ s\ \} \dashv \Gamma'}[\textsc{Method}]$$

Fig. 3: Methods.

$$\frac{\Gamma \vdash s_1 : \tau' \dashv \Gamma' \quad \Gamma' \vdash s_2 : \tau'' \dashv \Gamma''}{\Gamma \vdash s_1; s_2 \dashv \Gamma''}[\textsc{Stm}] \quad \frac{\Gamma \vdash e : \tau \dashv \Gamma'[e \mapsto \tau]}{\Gamma \vdash \texttt{return}\ e : \tau \dashv \Gamma'}[\textsc{Ret}]$$

$$\frac{\Gamma \vdash b \dashv \Gamma' \quad \Gamma' \vdash s \dashv \Gamma''}{\Gamma \vdash \texttt{while}\ (b)\ s\ \dashv \Gamma''}[\textsc{While}] \quad \frac{\Gamma \vdash e : \tau \dashv \Gamma'[e \mapsto \tau]}{\Gamma \vdash x = e : \tau \dashv \Gamma'[x \mapsto \tau]}[\textsc{Assign}]$$

$$\frac{\Gamma \vdash b \dashv \Gamma' \quad \Gamma' \vdash s_1 \dashv \Gamma'' \quad \Gamma' \vdash s_2 \dashv \Gamma'''}{\Gamma \vdash \texttt{if}\ (b)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \dashv \Gamma'' \cup \Gamma'''}[\textsc{If}]$$

Fig. 4: Statements.

Methods m**.** We elide many details of the definition of a program and we represent it just as a list of methods. To simplify the exposition, we remove all the information regarding packaging, visibility, type hierarchy, and global variables. Nevertheless, the details of how our approach handles a Java class are briefly presented in Section 3.3. Each method m is composed of an identifier i, a list of variables $(x_1, \_, x_n)$ that represent its input parameters and a sequence of statements s.

Statements s**.** We include here the syntax for the assignment, if, while, and return statements. The other conditional and loop statements typically provided in programming languages are handled in a similar way. The rule for the assignment statement assigns the expression e to variable x. The rules for the if and while statements contain a boolean expression b. Concerning the if statement, if b is true, the sequence of statements s_1 is executed otherwise s_2. Concerning the while statement, if b is true, the sequence of statements s is executed. The rule for the return statement returns the expression e.

Expressions e**.** Regarding literals n, the rule considers only integer values. Furthermore, our syntax supports method calls in the form $obj.m(e_1, \_, e_n)$, the basic mathematical operations, and min/max operations.

Booleans b**.** We support all the boolean operators for the comparison of timestamps. Note that equality checks do not suffer from overflow comparison problems and therefore, we skip them. We also include the logical conjunction and disjunction of boolean expressions.

3.3 Type Inference Rules

Our approach accepts as input a well-typed program and outputs the program annotated with the time type information. The type inference rules describe how TTS assigns a time type to literals, variables, and expressions. The rules are expressed via operational semantics [24] and they consist of a set of premises and a conclusion. Both premises and the conclusion are judgments. A judgment has the form $e : \tau$ which means e has type τ. In the context of the paper, τ refers to a time type. Judgments include a type environment Γ that contains the set of type bindings from variables and expressions to their respective time types. Since assignments can change the binding of a variable to a new time type, we designed our type system to be *flow-sensitive* which is achieved by inserting an output environment Γ' in addition to the input environment. For the sake of readability, we

$$\Gamma_0 := \Gamma \quad \forall_{i=1}^{n} \frac{\Gamma_{i-1} \vdash e_i : \tau \dashv \Gamma_i}{\Gamma_{i-1} \vdash e_i : \tau \dashv \Gamma_i[e_i \mapsto \tau]}$$

$$\frac{\Gamma'_0 := \Gamma_n \quad \forall_{(i,\tau) \in PosType(m)}\ \frac{\Gamma'_{i-1} \vdash e_i : \tau \dashv \Gamma'_i[e_i \mapsto \tau]}{}}{\Gamma \vdash obj.m(e_1, \ldots, e_n) \dashv \Gamma'_n}[\textsc{Et}]$$

$$\frac{m \in RT_\tau \quad \Gamma \vdash obj.m(e_1, \ldots, e_n) \dashv \Gamma'}{\Gamma \vdash obj.m(e_1, \ldots, e_n) : \tau \dashv \Gamma'}[\textsc{Rt}] \quad \frac{n \in \texttt{int} \vee n \in \texttt{long}}{\Gamma \vdash n : D \dashv \Gamma}[\textsc{Literal}]$$

$$\frac{\Gamma \vdash e_1 : \tau \dashv \Gamma' \quad \Gamma' \vdash e_2 : \tau \dashv \Gamma''}{\Gamma \vdash \texttt{min}\ \texttt{max}(e_1, e_2) : \tau \dashv \Gamma''}[\textsc{Min-Max}] \quad \frac{x \in \Gamma \quad \tau := \Gamma(x)}{\Gamma \vdash x : \tau \dashv \Gamma}[\textsc{Var}]$$

$$\frac{\begin{array}{ll} \Gamma \vdash & e_1 : \tau' \quad \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau'' \quad \dashv \Gamma'' \end{array} \quad \begin{cases} ① \vee ② & : \odot = - \\ ② \vee ③ \vee ④ & : \odot = + \\ ④ & : \odot = \times \\ ④ & : \odot = \div \end{cases}}{\Gamma \vdash e_1 \odot e_2 : \tau_0 \dashv \Gamma''}[\textsc{Int}]$$

$① : \tau' = \tau'' \rightarrow \tau_0 := D \qquad ② : \tau' = T \wedge \tau'' = D \rightarrow \tau_0 := T$
$③ : \tau' = D \wedge \tau'' = T \rightarrow \tau_0 := T \qquad ④ : \tau' = D \wedge \tau'' = D \rightarrow \tau_0 := D$

Fig. 5: Expressions.

$$\frac{\Gamma_0 = \Gamma \quad \forall_{i=1}^{n} \Gamma_{i-1} \vdash b_i \dashv \Gamma_i \quad \odot \in \{||,\ \&\&\}}{\Gamma \vdash b_1 \odot \cdots \odot b_n \dashv \Gamma_n}[\textsc{Bool}]$$

$$\frac{\begin{array}{ll} \Gamma \vdash & e_1 : \tau \quad \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau \quad \dashv \Gamma'' \end{array} \quad \odot \in \{<, <=, >=, >\}}{\Gamma \vdash e_1 \odot e_2 \dashv \Gamma''}[\textsc{Comp}]$$

Fig. 6: Booleans.

have shortened the names of time types Timestamp and Duration with T and D, respectively.

Example. The following shows an example of a typing rule consisting of two premises e_1 and e_2. The rule is read as: given that e_1 has type Duration and e_2 has type Timestamp in the type environment Γ, then the sum of the expressions e_1 and e_2 has type Timestamp. Therefore, variable x has type Timestamp in the output environment Γ'.

$$\frac{\Gamma \vdash e_1 : D \dashv \Gamma \quad \Gamma \vdash e_2 : T \dashv \Gamma}{\Gamma \vdash x = e_1 + e_2 : T \dashv \Gamma[x \mapsto T]}$$

The reasoning of the rule is based on the notion that if we add some time to a date, the result is still a date but in the future. Moreover, since the expression is assigned to variable x, the resulting environment extends Γ with a new mapping between x and its time type T.

Figure 3 presents the starting point of our analysis. The extended time semantics could detect that a class attribute or a parameter of a method is time related. In fact, the function $isTimeVar$ returns *true* if the expression in input is a reference to a time variable. However, it is not possible to know a priori the exact time type of the time parameters/attributes because they do not always have an initialization expression that can be used to infer their time type. Therefore, we introduce the **DoT** time type which expresses that a time expression is of type Duration or Timestamp. Time parameters of a method and time attributes of a class are initialized in the environment with the type DoT as defined in rule Method by the environment Γ_0. Through the analysis of the code, TTS tries to infer their precise time types, choosing between Duration and Timestamp.

$$\frac{\Gamma \vdash e_1 : \tau' \dashv \Gamma' \quad \Gamma' \vdash e_2 : \tau'' \dashv \Gamma''}{\Gamma \vdash \texttt{min}/\texttt{max}(e_1, e_2) : \tau_0 \dashv \Gamma''} \; \tau_0 := \begin{cases} \tau' & \tau' \neq DoT \wedge \tau'' = DoT \\ \tau'' & \tau'' \neq DoT \wedge \tau' = DoT \\ DoT & otherwise \end{cases} \; [\text{DoT-Min-Max}]$$

$$\frac{\begin{array}{lll} \Gamma \vdash & e_1 : \tau' & \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau'' & \dashv \Gamma'' \end{array} \quad \begin{cases} ① \vee ② \vee ③ \vee ④ & : \odot = - \\ ④ \vee ⑤ \vee ⑥ \vee ⑦ & : \odot = + \\ ② \vee ⑧ & : \odot = \times \\ ② \vee ⑧ & : \odot = \div \end{cases}}{\Gamma \vdash e_1 \odot e_2 : \tau_0 \dashv \Gamma''} \; [\text{DoT-Int}]$$

① $: \tau' = T \wedge \tau'' = DoT \rightarrow \tau_0 := DoT$ ② $: \tau' = D \wedge \tau'' = DoT \rightarrow \tau_0 := D$
③ $: \tau' = DoT \wedge \tau'' = T \rightarrow \tau_0 := D$ ④ $: \tau' = DoT \wedge \tau'' = D \rightarrow \tau_0 := DoT$
⑤ $: \tau' = T \wedge \tau'' = DoT \rightarrow \tau_0 := T$ ⑥ $: \tau' = D \wedge \tau'' = DoT \rightarrow \tau_0 := DoT$
⑦ $: \tau' = DoT \wedge \tau'' = T \rightarrow \tau_0 := T$ ⑧ $: \tau' = DoT \wedge \tau'' = D \rightarrow \tau_0 := D$

Fig. 7: DoT Expression Type Inference.

Figure 4 presents the rules for handling basic statements. Rule Stm shows how a sequence of statements is processed one after the other. Rule Ret unpacks the expression of a return statement and analyzes it recursively. Conditional and cyclic execution statements may have a timestamp comparison in their guard. Rules If and While first process the guard and then each of their sequence of statements independently. Finally, rule Assign updates the environment adding (or updating) the time type of the variable x. Variable declarations are handled as assignment expressions.

Figure 5 shows the rules for handling expressions. For every method call TTS verifies if it belongs to a time method of the extended time semantics. If the called method contains a time parameter in its signature, rule Et processes each argument recursively and updates the type environment accordingly. The rule uses an auxiliary function $posType$ that returns for the method m the set of positions for its arguments that are expected to be time related and their expected time types. If the call is to a method that returns a time value, rule Rt returns the time type of the expression performing a look-up in the sets of time methods. Furthermore, TTS considers every numeric literal that appears in an expression as Duration type. The time type system supports the most common mathematical operators that can be used with the `integer` types described by the Int rule. Depending on the operator used, we consider four different cases. TTS is designed to accept only *correct* operations between time values. In fact, operations such as the sum between two timestamp values is not accepted because the sum of two dates is not a meaningful operation. The subtraction operator, instead, accepts two timestamp values and the operation results in a Duration time type. Moreover, TTS supports also the functions `max` and `min`. Developers tend to use such functions to sanitize the input of time variables [21]. These functions return a time type equal to the time types of their arguments. Finally, rule Var performs a lookup in the type environment for a time variable and its time type.

The rules for handling boolean expressions are presented in Figure 6. They do not return any time type but their expressions can modify the environment and they are the code locations where timestamp comparisons appear. Rule Bool shows that each boolean expression connected by a boolean operator is processed left-to-right following the Java specification [25]. Expressions that compare time, as depicted by rule Comp, must have the same time type on both sides of the comparison.

The remaining rules describe how TTS handles DoT expressions and they are presented in Figure 7. The general idea of these rules is to infer the precise[5] type whenever it is possible. For instance, in rule DoT-Min-Max if the first argument is of type DoT and the second argument is of Duration, TTS infers that the first argument is of type Duration and it updates the type environment accordingly. The update of the environment is performed by analyzing the expression and obtaining the correct type for all the DoT time variables through the unification via pattern matching [26] with the cases of the rules that do not accept the DoT time type.

Example. Consider the following piece of code:

```
1  void foo(long time){
2    //{time : DoT}
3    long now = System.nanoTime();
4    //{time : DoT, now : T}
5    long tmp = (time - now) * 1000;
6    //{time : T, now : T, tmp : D, 1000 : D}
7    Thread.sleep(tmp);
8    //{time : T, now : T, tmp : D, 1000 : D}
9  }
```

Through the application of our time semantics, the parameter `time` is marked as a time variable. Through the rule Method, the type environment contains an entry for this parameter mapping it to the type DoT. The first statement in Line 3 matches the rule Assign and the right-hand-side of the assignment the rule Rt. Through the extended time semantics, TTS infers that the method call has type Timestamp because its signature is in RT_t. Thus, it adds an entry to the type environment for variable `now` mapping it to the type Timestamp. Analyzing the right-hand-side of the assignment statement in Line 5, rule Int matches the multiplication case. The Int rule then analyzes both operands where the second operand, being a scalar value, has type Duration. The first operand, instead, is a subtraction referencing the variable `time` that has type DoT. Therefore, rule DoT-Int matches with ③ of the first case assigning the subtraction the type Duration. Since there is a match of a DoT rule, TTS tries to infer the appropriate time type for the `time` parameter. Currently, TTS has the expression $DoT - T = D$ that it tries to match with the cases of the rule Int. This rule has two cases for the subtraction operator, but the latter does not match because it requires a Timestamp type on the right-hand side instead of a Duration. Then, ① of rule Int is the only case that matches and therefore, TTS infers that the parameter `time` has type Timestamp.

Unfortunately, it is not always possible to infer a concrete type for DoT variables. The following code presents an example of such a case:

```
1  long fee(long time){
2    //{time : DoT}
3    long ms = TimeUnit.toMillis(100);
4    //{time : DoT, ms : D}
5    return (time - ms);
6    //{time : DoT, ms : D, return : DoT}
7  }
```

The environment is initialized with the parameter `time` of DoT time type. After the first statement, the environment

5. Either Timestamp or Duration

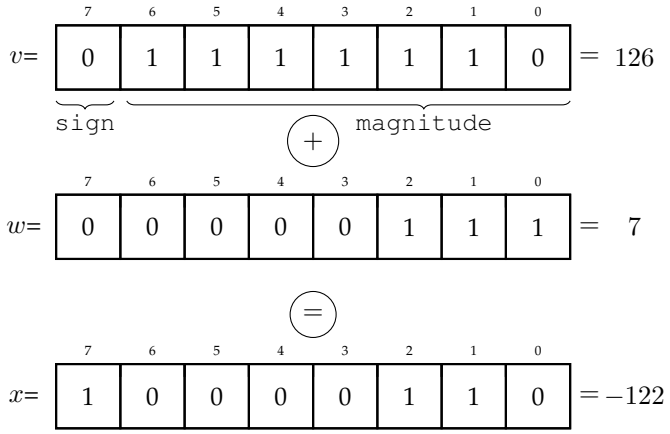

Fig. 8: Example of an integer overflow due to the two's complement representation.

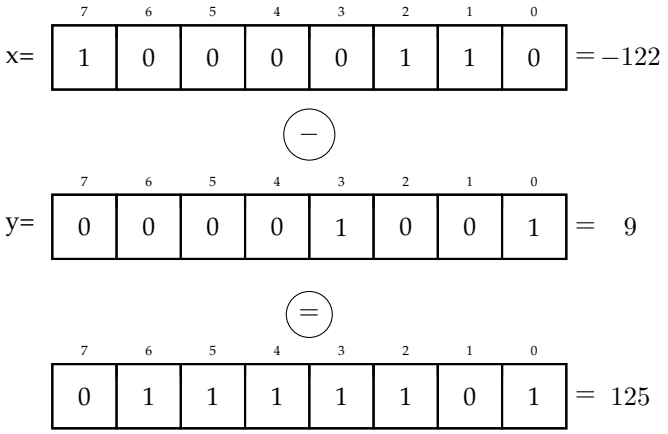

Fig. 9: Example of how to exploit the two's complement representation to cancel an integer overflow.

contains the information that variable `ms` has type Duration. Analyzing the expression of the return statement, the time type system matches rule DoT-Int with ④ of the first case. When TTS tries to infer the correct type for variable `time` using the cases of rule Int, both cases of the subtraction match. Since there is no unique case that matches, TTS is not able to infer the correct type because both Timestamp and Duration are valid types for this expression.

4 Repairing Timestamp Comparison

Looking at the example presented in Listing 1, the condition `currTimeMs >= endTimeMs` might wrongly throw a `TimeoutException` because the value for the variable `endTimeMs` might overflow. Such a comparison is therefore called *linearly unstable* according to the linear stability principle [27]. The reason lies in how the Java Virtual Machine (JVM) handles overflows and arithmetic operations. Java uses the standard representation of two's complement [28] to represent integer numbers. The first bit in a two's complement representation indicates the sign of the number, where 1 represents a negative and 0 a positive number. The part from the second bit until the end of the binary representation is called *magnitude*. Figure 8 shows an example of an integer overflow in the two's complement representation. Variables v, w, x use 8 bits to represent integer numbers in the range from -128 to 127. The sum of v (126) and w (7) causes an integer overflow resulting in -122 stored to x. The two's complement representation is therefore not linearly stable because a small perturbation on the input value can lead to an unexpected large change in the result. In our example, the sum of 7 and 126 leads to -122 that is way different from the expected result, namely 133.

As argued in the introduction, the integer values of timestamps are typically large so that mathematical operations, such as the plus-operation, while syntactically correct, are prone to integer overflows. Regarding the example presented in Listing 1, we can express the comparison with a mathematical equivalent but more stable expression: `endTimeMs - currTimeMs <= 0`. In terms of our time type system, our approach rewrites the comparison between two Timestamps to a comparison between two Durations while preserving its semantics. Although mathematically equivalent, the two expressions differ due to the two's complement representation used by computers [29]. If one of the two Timestamps suffered from an overflow, the subtraction will likely lead to an integer underflow that cancels the overflow. Figure 9 continues the example presented in Figure 8. If after the overflow stored in x, there is a comparison such as `if (x < y)`, the statement will return `true` ($-122 < 9$), although the programmer expects it to be `false` because the expected comparison $133 < 9$ is false. On the contrary, if the code contains the condition rewritten as `x - y < 0`, it will return, as expected, `false` because the subtraction $-122 - 9$ results in an integer underflow, namely 125, that compensates for the previous overflow. The condition $125 < 0$ then returns, as expected, `false`.

We call *normal form* for Timestamp comparisons, a comparison in the form:

$$expr_1 - expr_2 < 0 \tag{1}$$

Then, we can express the problem of rewriting the comparison between two Timestamp expressions as a refactoring problem that improves the program's design, while preserving its current behavior [30], [31]. The behavior of a program is specified by the developers or it can be obtained from the program itself through the execution of the test suite. However, tests may not give the complete picture, therefore we call the behavior extracted through the execution of the tests the *observable behavior*. In our approach, we alter the program's behavior only for the currently unobserved behavior that contains errors. Note that, we only present the case for the $<$ operator. Other relational operators that check for inequality, *i.e.*, $\leq, >, \geq$, follow analogously the approach.

A common technique to specify a refactoring problem is using Hoare triples [32] that are composed of three parts: precondition, procedure, and postcondition. We define our refactoring problem with a defensive implementation [33] using the following Hoare triple:

$$\begin{gathered} true \\ \{\texttt{if pre}(\mathcal{P})\ \texttt{then change}(\mathcal{P})\} \\ (Pre(\mathcal{P}) \wedge Post(\mathcal{P}, \mathcal{P}')) \vee (\neg Pre(\mathcal{P}) \wedge \mathcal{P}' = \mathcal{P}), \end{gathered} \tag{2}$$

where $\mathcal{P}'$ refers to the result of the refactoring procedure over the program $\mathcal{P}$; $Pre(\mathcal{P})$ is the function that protects the refactoring from generating a wrong program rejecting every program that the refactoring specification cannot handle; and $Post(\mathcal{P}, \mathcal{P}')$ is the function that validates the correctness of the refactoring procedure that is applied to the program $\mathcal{P}$; $\texttt{pre}(\mathcal{P})$ is the implementation of the function $Pre(\mathcal{P})$; and $\texttt{change}(\mathcal{P})$ is the implementation of the function that rewrites any Timestamp comparison into the normal form. The precondition in our case is true, because the validity of the application of the refactoring procedure is part of the action that is implemented by the refactoring procedure in the refactoring tool. The postcondition ensures that only if the application of the refactoring procedure is possible, the output contains the correct changes.

$$Post(\mathcal{P}, \mathcal{P}') = wellFormed(\mathcal{P}) \wedge wellFormed(\mathcal{P}') \wedge obsBehavior(\mathcal{P}) = obsBehavior(\mathcal{P}') \quad (3)$$

Equation 3 shows how the refactoring procedure can be validated. The output program $\mathcal{P}'$ should exhibit the same observable behavior of the input program and should be well formed, *i.e.*, it does not introduce any syntactical error and it can successfully compile and execute the test suite.

Based on Hoare triples, the above formalization defines the necessary conditions under which a refactoring tool correctly changes a program. However, different refactoring (and repairing) tools can be developed by realizing different implementations of the functions $Pre(\mathcal{P})$ and $\texttt{change}(\mathcal{P})$. Although easy to express, the function $Pre(\mathcal{P})$ and its implementation $\texttt{pre}(\mathcal{P})$ are in practice a major problem, such as demonstrated in [34], [35], [36], [37], [38], [39], [40], [41], [42]. For our purpose, $Pre(\mathcal{P})$ requires that the comparison is performed between two expressions that have type Timestamp. We use TTS to check the precondition of the problem. If fulfilled, which means TTS identified a comparison between two Timestamp expressions, our approach refactors it by rewriting the comparison expression into its normal form. Finally, our approach verifies the post-condition using the build system of a project: first it verifies that the modified program compiles and is well-typed ($wellFormed$); second, it executes the test suite to verify that the refactoring did not change the observed behavior.

5 Experiments

In this section, we present the experiments we have performed to evaluate our approach to repair timestamp comparisons. We address the following three research questions:

- **RQ1:** What is the precision and recall of TTS in inferring time types?
- **RQ2:** What is the correctness of the patches created by our approach?
- **RQ3:** What is the usefulness of the patches created by our approach?

The next subsections describe the setup of the experiments and how we performed them.

5.1 Setup

We have implemented our approach in an open source prototype tool used to answer the three research questions. For the dataset creation, we queried GitHub for Java projects and we ranked them based on their number of stars. We manually filtered the results retaining projects that use distributed components or communication protocols over the network to enhance the likelihood of timestamp comparisons. From the resulting list, we randomly selected 5 projects that have less than 1500 stars, 10 projects that have between 1500 and 5000 stars, and 5 projects that have more than 5000 stars. Our search resulted in a dataset comprising 20 Java projects that vary in size, vendor, and domain.

For each project, we downloaded the source code of the latest release. Table 1 lists the 20 projects together with descriptive statistics of their size, number of tests, and statement coverage. We have verified that each project is configured for reporting the statement coverage. If no coverage support was defined, we added the Cobertura[6] plugin into the build script using its default configuration to compute the statement coverage score. Regarding the projects size, the number of classes per project varies from 124 to $27,208$ containing from 716 up to $205,432$ methods that in total comprise $9,590,951$ SLOC. The number of tests per project varies from 162 up to $150,648$ resulting in a total of $302,616$ tests. Regarding the coverage, Airavata is the project with the lowest coverage, namely 8.25%. Elasticjob has the highest coverage of 87.84%. On average, the provided tests cover 51.93% of the projects' source code.

Our prototype tool analyzed the $9,590,951$ SLOC in 11.72 hours.[7] The AWS project alone took most of the time, namely 7 hours. Although AWS is 68% bigger than Camel in terms of SLOC, it required 86% more time to compute. This means that the run time required by our prototype is not linearly increasing with the size of a project. In fact, it depends on how the source code is structured. We observed that the AWS project has many if-then (and switch) statements that are expensive operations for TTS: when analyzing a branching instruction, TTS creates a new copy of the environment for each branch which is a time and memory consuming operation.

5.2 RQ1 - Identifying Timestamp Expressions

With the first experiment, we want to investigate if the definition of TTS is adequate to correctly infer the time types of expressions. Since TTS can be proved to be sound but not complete, we studied its empirical precision and recall.

To have an estimate value for the recall, we ran our prototype tool on the source code of each project applying the time semantics to identify the time variables that appears on the left-hand-side of an assignment expression. Then, we used TTS to compute their time types. We counted the number of time variables for which TTS inferred a time type (Duration or Timestamp) and the number of variables for which it could not (they have type DoT). The results of this experiment are presented in the columns #Time Var, #Typed, and #DoT of Table 1. Over all projects, TTS identified $1,069,598$ time variables and for most of them, namely $1,069,265$, it infers a time type, *i.e.*, either D or T. Only for 333 time variables (0.03%) it could not infer

6. http://cobertura.github.io/cobertura/

7. All experiments have been conducted on a computer with a 2.5 GHz Intel CPU and 16 GB of physical memory running macOs 10.13.5.

TABLE 1: Results of the application of our approach on the 20 Java projects showing the short commit hash (# Hash), classes (# Classes), methods (# Methods), test and their coverage (# Tests), single line of code (SLOC), the seconds required for the analysis (Time), the number of time variables identified (# Time Var), the number of time variables that are assigned with a Time Type (# Typed), the number of time variables for which TTS was not able to decide between Duration or Timestamp time type (# DoT), and finally the number of patches produced (# Patches).

Name	# Hash	# Classes	# Methods	# Tests	SLOC	Time [s]	# Time Var	# Typed	# DoT	# Patches
Activemq	ccf56875b	5,100	44,072	20,450 (30.41%)	421,839	473.63	31,094	31,071	23	23
Activiti	917246113	2,103	15,381	3,968 (60.68%)	139,672	203.48	11,266	11,266	0	0
Airavata	391843a00	9,320	70,875	162 (8.25%)	711,587	1,146.17	131,439	131,439	0	1
Alluxio	2bf790f505	3,364	24,975	4,270 (45.85%)	233,897	223.00	36,854	36,833	21	8
Atmosphere	d51726dcc	500	4,101	504 (55.16%)	35,843	35.86	4,074	4,072	2	0
Aws-sdk-java	5984638d0b	27,208	205,432	2,586 (57.33%)	1,795,234	28,091.57	186,336	186,335	1	9
Beam	3b03106b55	3,844	21,404	8,930 (66.64%)	210,960	201.79	19,548	19,543	5	1
Camel	497fa7760e1	20,024	116,080	47,704 (40.68%)	1,065,292	4,696.54	77,766	77,760	6	6
Elastic-job	dbb79ef4	611	2,497	1,842 (87.84%)	26,418	19.17	1,942	1,941	1	0
Flume	7d3396f2	995	6,705	2,288 (48.00%)	85,750	52.29	9,086	9,075	11	8
Hadoop	128dd91e100	12,597	100,635	10,830 (51.16%)	1,267,414	1,955.95	121,859	121,715	144	35
Hazelcast	02e3fbf737	7,663	59,294	11,035 (76.57%)	649,789	736.54	37,383	37,355	28	1
Hbase	44f8abd5c6	9,535	128,928	4,614 (34.11%)	1,201,149	1,995.11	248,935	248,895	40	51
Jetty	65528f76c5	3,781	25,554	12,742 (45.65%)	342,602	301.51	26,568	26,559	9	8
Kafka	c74acb24e	1,896	14,007	9,331 (71.87%)	149,644	119.29	16,440	16,429	11	19
Lens	cdd7b099	1,036	8,114	2,432 (53.83%)	99,523	75.88	10,622	10,613	9	9
Nanohttpd	f1cb85c	124	716	478 (75.25%)	7,532	4.25	742	742	0	0
Neo4j	a41464ed3ba	9,158	61,407	150,648 (53.61%)	680,986	770.02	43,814	43,804	10	18
Sling	73fe13fa28	6,022	38,049	7,078 (35.80%)	433,384	1,046.44	50,845	50,834	11	49
Twitter4j	cf6afc3e	418	4,642	724 (40.00%)	32,436	41.63	2,985	2,984	1	0
SUM	**-**	**125,299**	**952,868**	**302,616 (51.93%)**	**9,590,951**	**42,190.12**	**1,069,598**	**1,069,265**	**333**	**246**

the time type, resulting in an overall recall of 99.97%. For instance, in HBase $248,895$ time variables were typed and only for 40 time variables, TTS was not able to infer their time types. For Airavata, TTS inferred for all the $131,439$ time variables their time types.

Regarding the precision, we randomly selected 400 out of the $1,069,265$ time variables to obtain results with 95% level of confidence and 5% margin of error. For these time variables, we manually assessed their time types based on the assignment expression. Then, we ran TTS and verified that its output is aligned with the manual results. The manual analysis was performed by the authors of the paper and by an external developer with an inspection of the source code that contains each of the 400 selected time variables. The manual inspection was performed using the IntelliJ[8] editor and its slicing and point to features to navigate the code. They first analyzed the expression that assigns a value to the time variable and then used IntelliJ to backward slice the code to assess that the correct type was indeed inferred w.r.t. the classification of time method of the Java APIs (see Section 3.1). Analyzing the 400 time variables, 309 (77.25%) were manually categorized with Timestamp type and 91 (22.75%) with Duration type. For all the variables, TTS computed the same time type obtained with the manual analysis. Based on this result, we conclude that TTS has an empirical precision of 100% in inferring time types.

5.3 RQ2 - Patch Correctness

For ActiveMQ, Atmosphere, Elastic-Job, NanoHttpd, and Twitter4j our prototype tool was not able to find errors and create a patch to repair them. However, for the other 15 projects it was able to propose 246 patches. From the previous research question, we know that TTS infers correctly the time types of expressions. To further verify this, we manually assessed, using the aforementioned methodology, that each side of all faulty comparisons identified by TTS is indeed a Timestamp expression. We confirmed that all the 246 comparisons to repair are between Timestamps expressions and they represent the input for our second evaluation.

In this evaluation, we investigated if the repair preserves the program semantics while removing potential overflow errors. According to the problem definition presented in Equation 3, the post condition assures that the repaired program is well formed and the observable behavior is maintained *i.e.*, the patch does not introduce any new error in the program. To evaluate if our patches do not break the post conditions, we applied each proposed patch one at a time. After each patch application, we ran the test phase of the build system of the project to verify that the post condition for the repaired program holds. We ran the tests without any additional configuration or parameter. We used the report of Cobertura to confirm that each repaired statement was executed by at least one test. The build system first, assures that the program is well formed through the compilation of the source files and second, that the observable behavior is maintained through the execution of the test suite. For all the 246 patches, the build system executed the $302,616$ tests without any error. This shows that our approach correctly repairs the 246 errors without introducing any side effect that can alter the program's behavior in other parts of the system.

5.4 RQ3 - Patch Usefulness

In addition to the previous assessments, we also studied the usefulness of the patches in two different ways. First, the first author of the paper, an independent researcher, and an external developer, who have several years of academic and professional experience in developing Java applications, verified through a manual control- and data-flow analysis if the Timestamp comparisons identified by TTS are indeed subjected to overflow errors. Second, we applied the patches to the projects and submitted them as pull-requests to obtain the feedback from the original developers of the projects.

8. https://www.jetbrains.com/idea/

The manual data- and control-flow analysis has been performed independently by each researcher and by the developer, analyzing the faulty comparisons identified by TTS to assess that they required to be repaired. The analysis has been performed in the same manner as presented in Section 5.2. Then, all participants came together and discussed the results. For each discrepancy in their results, they analyzed together the code to reach a consensus agreement. They have discovered that 41 of the 246 repaired time comparisons are not suffering from errors due to overflow. Therefore, our fix is not strictly necessary. In the 41 cases, the developers handle the potential overflow in the source code. For instance, TTS identified the following problematic if-condition in the Hadoop project:

```
if (cacheExpiryTimeStamp >= 0 &&
    cacheExpiryTimeStamp < now)
```

Variable `now` holds the current system clock value and therefore it is certain that the variable always stores a correct, *i.e.*, not overflowed, time value. Variable `cacheExpiryTimeStamp`, instead, holds the result of a mathematical operation on timestamps that could overflow. The developers of Hadoop have protected the code against the overflow by adding the condition `cacheExpiryTimeStamp >= 0` to the if-condition. The overflow itself then is handled in the else-branch.

Regarding the evaluation with developers, we selected three projects namely, ActiveMQ, Alluxio, and Kafka. The restriction to these three projects was necessary because, for getting pull-requests accepted, it is mandatory to create a test harness for every failure detected. The creation of the test harness is time consuming since it requires to be familiar with the projects and their source code. At the moment of writing we were able to create and submit 3 pull-requests including the test-harnesses to the three projects. In total, the 3 pull-requests fix 24 errors in timestamp comparisons. We obtained feedback on all our pull-requests that we used to evaluate the usefulness of our patches. The developers of ActiveMQ responded with *"Good catch!"* and the ones from Alluxio with *"Really cool fix!"*. In both projects, they accepted our patches and merged the pull-requests.[3,9] In the Kafka project, instead, our pull-request[10,11] started a discussion on how to best handle timestamp overflows. While they acknowledged our solution, they also found that a more readable and maintainable solution to this issue is needed. They suggested to rewrite the logic of the program, if possible, avoiding mathematical operations that could lead to an overflow. For the moment, however, the developers decided to accept our pull-request and to roll-out the better fix in the next releases. Furthermore, another developer of the project proposed[12] to centralize the handling of time manipulation with a specific class that *correctly* implements all the logic necessary to modify and compare time values.

Based on these results we can answer research question RQ3 as follows: our approach showed evidence that it can aid developers to repair Timestamp comparison and it can be integrated into the deployment pipeline where developers can push unsafe code that is automatically repaired.

9. https://github.com/apache/activemq/pull/284
10. https://github.com/apache/kafka/pull/5078
11. https://github.com/apache/kafka/pull/5183
12. https://github.com/apache/kafka/pull/5087

6 Discussion

In this section, we discuss the outcome of our evaluation and its implication for researchers and practitioners. Furthermore, we discuss the limitations of our approach and the potential threats to validity of our empirical studies.

6.1 Summary of Results

With our three research questions, we studied three aspects of our approach. The first research question is designed to investigate the ability of our proposed time type system to identify Timestamp comparisons. Since identifying comparisons is trivial, we studied the ability of TTS in inferring the correct time type for source code expressions. The results show that among the 20 projects, it is able to infer a time type with a recall of 99.97% and a precision of 100%. The second and third research questions, instead, are designed to investigate the repairing correctness and usefulness of our approach. First, we assessed that the patches are correctly repairing the errors without introducing new ones. Second, we published our 24 repairs as 3 pull-requests in the project repositories to get the feedback from the developers. Our patches have been applied to three different versions of Apache ActiveMQ and after the discussion in our pull-request, the developers of Apache Kafka have been working on refactoring the full system to change how timestamp comparisons are performed. This confirms the usefulness of our approach.

6.2 Implication of Results

Concerning the implications on the research in this area, the definition of a time type system opens a possible further area of research. We envision researchers using the definition of our time type system to study the evolution of time APIs and how they are refactored or changed by evolving the software project. In fact, changing the Hoare triple that defines the problem, our approach can target different problems related to time properties. Furthermore, existing taxonomies and studies on code smells and anti-patterns can be extended by considering time types.

Our solution can be applied to any language that uses the two's complement representation. We use Java only as case study to show the applicability of our approach. Therefore, researchers on compilers definition and development could integrate the ideas of TTS and directly provide out-of-the-box better basic types to express time values in the program. This would improve static checkers that can be run on the source code before producing the machine code. Furthermore, TTS can be used to identify semantic errors in time related expressions. For instance, a multiplication between Timestamp values is a valid Java operation but it constitutes an error that is identified by TTS. However, we conjecture that in our experimental validation, we never encountered such errors because we addressed mature projects. The investigation of the usefulness of TTS inside an IDE to help developers identify such errors while coding is left for future studies. Our results have also several implications for practitioners. Developers can use our approach in their continuous integration and continuous delivery pipeline. When a commit is pushed into the version

control system, our approach can process the commit and automatically apply the transformation on timestamp comparisons. If it is too expensive to apply for every commit, they could introduce it only in the nightly or weekly build. Moreover, it can help developers discuss the logic of their system and consider to refactor the code as it happened with the developers of Apache Kafka.

6.3 Limitations

In our approach, we present a static analysis technique that infers time types for time expressions. In principle, our time type system is sound, which implies that it assures the typical timed-"nonstuckness" property, *i.e.*, any well-typed program cannot get stuck w.r.t. time. In other words, it establishes the fact that the time variables, as determined by the typing rules defined in Section 3.3, contain time values. The proof of the soundness follows from the given typing rules based on the time semantics of Java [23]. The detailed proof is discussed in [43]. However, our approach is not complete, *i.e.*, it cannot avoid all overflow errors due to the finite arithmetic of computers. If the result of an expression is too big to be stored into the finite amount of bits available, there is no sort of the operations that will prevent the overflow. Nevertheless, we have investigated empirically via means of precision and recall the theoretical limitation of our approach. The result shows that, it can infer the time types of expressions with a precision of 100% and a recall of 99.97%. Another limitation of our approach is a lack of data-flow analysis. In fact, we repair 41 Timestamp comparisons that are not strictly necessary because developers already thought about a possible overflow and handled it with ad-hoc code. Even though it is not strictly necessary, the repair does not affect the correctness of the program but it introduces an extra operation to perform. Further studies are necessary to evaluate how these repairs are seen by developers.

6.4 Threat 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. One threat to the internal validity concerns the reliability of the prototype. We mitigated this threat formalizing TTS and proving its soundness to assure that the approach is designed correctly. Moreover, we mitigated errors in the implementation with manual and unit tests. Furthermore, we computed the precision of our approach with a manual analysis. Among the $1,069,598$ time variables that are assigned with time expressions, we randomly selected 400 that we have manually investigated. The size of our sample set exceeds the minimum number of 384 required to obtain results at a 95% confidence level with a 5% margin of error. Moreover, we mitigated possible threats to validity of the different manual evaluations performing the experiments independently by the different authors of the paper, the independent researcher, and the external developer.

In the first research question we based our computation of precision and recall on the number of time variables identified in the source code. Therefore, our computed values are valid w.r.t. the precision and recall of the defined formal time semantics. However, the formal time semantics has shown an empirical precision and recall of 100% in identifying time variables and time statements. This mitigates potential threats to the validity of our results.

Equation 3 shows that TTS works on the basis that the input program is well typed. In our studies we always used a version of the project that is compilable, *i.e.*, well typed, so the time type inferred are trustworthy. The assumption of having compilable source code is realistic since it is rare that a commit cannot be compiled [44]. Furthermore, we evaluated if the synthesized patches introduce any new error relying on the test suite of the projects. Therefore, if not enough tests are provided by developers there could be a chance that an unwanted side effect is generated by one of our patches and it passes unnoticed. However, this does not threaten our studies because we have proved in [43] that our repair strategy repairs possible time overflows without altering the program's intended behavior, *i.e.*, the refined expression is formally proved to be semantically equivalent to the original expression.

External Validity. Threats to the external validity of our studies concern the generalization of the results. We mitigated this threat by choosing open source Java projects that differ in vendor, size, and domains. Moreover, we implemented our approach in a prototype tool that is publicly available online.[4] Other researches can freely use our tool and apply it to other case studies and extend our results. Furthermore, our approach is based on how machines encode integer numbers with the two's complement representation. Therefore, our technique can be extended to other programming languages, such as C and C#, that use the same encoding and a similar time semantics.

7 Related Work

The main contribution of this paper is a technique that automatically repairs programs. In this domain, multiple researches address this problem with a generate-and-validate approach. When a test fails, the repairing strategy generates a patch and then, it validates or refuses it based on the outcome of the test suite. This kind of repairing approaches rely on building the set of all possible changes, called search space, that can be applied to repair the defected code. Long *et al.* [6] studied how changing the search space for existing techniques, the repairing success rate changes. They discovered that including information taken from outside of the test suite enables these systems to successfully identify more correct patches. Le *et al.* [45] propose a genetic algorithm to repair defects. The algorithm can be run in a cloud environment with an average cost of 8$ per patch, solving 50% of the discovered faults. Instead, Nguyen *et al.* [3] propose an approach that creates a constraint problem from the source code based on the tests execution. Then, it generates a patch that satisfies such constraint problem. A common shortcoming of these techniques is that they overfit the problem and generate patches that pass the test cases rather than repairing the code. Recent researches start to use semantic information to repair the program without overfitting. Ke *et al.* [5] process a large corpus of projects to extract snippets of code that are encoded as SMT constraints

that are stored in a database. When a fault is discovered during the execution of a test, they derive the input-output relationship and query the database. The results of the query are used to synthesize a more accurate patch. Instead, Mechtaev *et al.* [8] present an approach that synthesizes a patch from a formal specification of the requirements of the project. Tonder and Le Goues [46], similarly to our approach, introduce a method that does not rely on testing to discover faults to repair. They model a program via an intermediate language that performs operation on a heap. Users can specify properties that must hold in the heap. Every time a property does not hold, their approach synthesizes a patch using the code where the property holds. However, this approach suffers from false positives due to its approximate analysis of loops and clean up functions. Furthermore, since a patch is generated from other code snippets, it might contain side effects that can introduce errors. In contrast, our approach is guaranteed to fix an error without altering the program semantics.

In the domain of overflow detection, Beckert and Schlager [47] propose the KeY specification for the Java language that combines the semantics of the infinite integer with the semantics of finite integer used by machines. Developers can write the specification of their software with KeY and then prove properties using Dynamic Logic. This work is suited to model check integer properties and it does not automatically identify or repair the program. Instead, Brumley *et al.* [48] present the definition of a type system that formally specifies the semantics for multiple undefined behaviors of the C99 language. The type system is used to identify integer expressions with undefined behavior and their approach inserts checks after them that detect integer attacks, such as integer under- and over-flows. Following a similar idea, Dietz *et al.* [49] present IOC, a tool that is part of the Clang toolchain to compile C/C++ programs on Mac OS and iOS. Their tool analyzes the LLVM [50] code representation of a C/C++ program and it identifies undefined behavior for integer operations. The main difference to our approach is that these approaches stop the program execution when integer errors occur without repairing the program. The work of Cocker and Hafiz [51] solves multiple integer overflow problems. They define three different operations that rewrite the source code of a C program to protect its runtime to suffer from integer problems. Their technique introduces and replaces common mathematical operations with calls to auxiliary library functions. The library correctly implements the mathematical operations and in case of overflow, it returns a runtime exception. Those transformations are security-oriented and thus, they broke the expected behavior of the program. This differs from our approach because we perform transformations that preserve the expected behavior of the program.

The closest related work is presented by Logozzo and Martel [52]. They proposed an approach that repairs integer overflows for arithmetics expressions. However, their work has some limitations. It works only on sum expressions and does not supports a full-fledged programming language. It cannot handle side-effects so modern programming languages cannot be targeted. Finally, it requires users input to specify templates for the expressions to repair. In contrast, our approach performs a full-fledged static analysis on the source code without requiring any user inputs.

8 Conclusion

In this paper, we presented a static analysis approach to automatically repair programs that does not rely on testing for discovering faulty code. Our approach repairs programs that suffer from problematic timestamp comparisons and we show its applicability for the Java programming language. We introduce a Time Type System (TTS) that is built on top of a time semantics of the programming language. TTS is used to identify the timestamp comparisons to repair. These comparisons are rewritten in a form that exploits how machines encode numeric values to produce a mathematical equivalent but more stable expression for comparing timestamp values. We performed three experiments on 20 open source projects to evaluate (i) the precision and recall of TTS in identifying timestamp comparisons, (ii) the correctness of the synthesized patches, and (iii) their usefulness for developers. The results show that our approach can identify timestamp comparisons with a precision of 100% and a recall of 99.97%. Furthermore, all the patches created correctly repair the 246 identified errors. We performed a manual analysis over the 246 patches and we discovered that for 41 of them, developers already knew the problem and handled it with ad-hoc code. We also published several patches for three of the 20 projects as pull-requests. All of them were acknowledged and accepted by the developers of the projects. Future work will be devoted to add a data-flow analysis to TTS to correctly find the errors that developers are aware of and are handled with ad-hoc code. Furthermore, our approach is general and can be applied to other programming languages. We plan to extend its support to more programming languages, such as C++ and C#. Moreover, we plan to study with an industrial partner how developers perceive our approach when it is integrated into their continuous integration and continuous delivery pipeline.

Acknowledgments

The authors would like to thank Veit Frick and Karin Hodnigg for reviewing the paper and the Austrian Research Promotion Agency FFG for funding this research within the FFG Bridge 1 program, grant no. 850757.

References

[1] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering (FOSE)*. ACM, 2000, pp. 73–87.

[2] R. N. Charette, "Why software fails [software failure]," *Ieee Spectrum*, vol. 42, no. 9, pp. 42–49, 2005.

[3] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.

[4] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 87–102.

[5] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 295–306.

[6] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 702–713.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 691–701.

[8] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of ICSE*. IEEE, 2018, pp. 129–139.

[9] C. Pacheco and M. D. Ernst, "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, 2007, pp. 815–816.

[10] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .NET with feedback-directed random testing," in *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA)*. ACM, 2008, pp. 87–96.

[11] M. Boshernitsan, R. Doong, and A. Savoia, "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, 2006, pp. 169–180.

[12] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[13] ——, "1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite," *Empirical Software Engineering*, vol. 20, no. 3, pp. 611–639, 2015.

[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[16] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, 2008.

[17] Facebook, "Infer static analyzer," 2017, accessed 05 Jun 2018. [Online]. Available: http://fbinfer.com/

[18] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, "Fast and precise type checking for JavaScript," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 48, 2017.

[19] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 580–586.

[20] G. Liva, M. T. Khan, and M. Pinzger, "Extracting timed automata from Java methods," in *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 91–100.

[21] G. Liva, M. T. Khan, F. Spegni, L. Spalazzi, A. Bollin, and M. Pinzger, "Modeling time in Java programs for automatic error detection," in *Proceedings of the IEEE/ACM Conference on Formal Methods in Software Engineering (FormaliSE 2018)*. IEEE Press, 2018.

[22] L. Spalazzi, F. Spegni, G. Liva, and M. Pinzger, "Towards model checking security of real time Java software," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 642–649.

[23] G. Liva, M. T. Khan, and M. Pinzger, "Semantics-driven extraction of timed automata from java programs," *Empirical Software Engineering*, pp. 1–37, 2019.

[24] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[25] Oracle. (2018) Java 8 language specification. Accessed 19 Jun 2018. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.7

[26] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[27] U. Itkis, *Control systems of variable structure*. Halsted Press, 1976.

[28] K. Hwang, *Computer arithmetic principles, architecture, and design*. John Wiley & Sons Inc, 1979.

[29] J. R. Rice, *Numerical Methods in Software and Analysis*. Elsevier, 2014.

[30] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, 1992.

[31] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, 1992.

[32] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[33] F. Steimann, "Constraint-based refactoring," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 40, no. 1, p. 2, 2018.

[34] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 185–194.

[35] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic testing of refactoring engines on real software projects," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 629–653.

[36] M. Schaefer and O. De Moor, "Specifying and implementing refactorings," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 286–301.

[37] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent Java code," in *European Conference on Object-Oriented Programming*. Springer, 2010, pp. 225–249.

[38] M. Schäfer, T. Ekman, and O. De Moor, "Sound and extensible renaming for Java," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 277–294, 2008.

[39] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor, "Stepping stones over the refactoring rubicon," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 369–393.

[40] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2013.

[41] F. Steimann and A. Thies, "From public to private to absent: Refactoring Java programs under constrained accessibility," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 419–443.

[42] F. Steimann and J. von Pilgrim, "Constraint-based refactoring with foresight," in *European Conference on Object-Oriented Programming*. Springer, 2012, pp. 535–559.

[43] G. Liva and M. T. Khan, "Proof of soundness," 2019. [Online]. Available: https://thisthat.github.io/papers/AAU-SERG-2019-001.pdf

[44] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[45] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 3–13.

[46] R. Van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *Proceedings of ICSE*. IEEE, 2018, pp. 151–162.

[47] B. Beckert and S. Schlager, "Software verification with integrated data type refinement for integer arithmetic," in *International Conference on Integrated Formal Methods*. Springer, 2004, pp. 207–226.

[48] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, and D. Song, "Rich: Automatically protecting against integer-based vulnerabilities," *Department of Electrical and Computing Engineering*, p. 28, 2007.

[49] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, p. 2, 2015.

[50] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[51] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 792–801.

[52] F. Logozzo and M. Martel, "Automatic repair of overflowing expressions with abstract interpretation," in *Electronic Proceedings in Theoretical Computer Science*, ser. EPTCS, vol. 129, 2013, pp. 341–357.

Giovanni Liva is working toward the doctoral degree at the Alpen-Adria Universität, Austria. He received his Computer Science B.Sc. in 2013 at Universitá degli Studi di Udine. In 2015 he received his M.Sc. degree cum laude in the joint program between Universitá degli Studi di Udine and Alpen-Adria-Universität Klagenfurt. His research interests include program analysis, abstract interpretation, model checking, and software evolution.

Muhammad Taimoor Khan Muhammad Taimoor Khan is Lecturer in Secure Systems at Surrey Center for Cyber Security, University of Surrey, UK. His current research focus is on developing secure by design cyber physical (robotic) systems. Since last decade, he has been applying formal methods to assure reliability and security of various software systems, e.g., industrial control systems, computer mathematics-based systems, to name a few. He has been working as a scientist at various premier international research institutes, including INRIA, France and MIT CSAIL, USA; he is jointly working with these institutes now. Dr. Khan has won various research and academic awards including best paper award(s).

Martin Pinzger Martin Pinzger is a full professor at the University of Klagenfurt, Austria where he is heading the Software Engineering Research Group. His research interests are in software evolution, mining software repositories, program analysis, software visualization, and automating software engineering tasks. He is a member of ACM and a senior member of IEEE.

Francesco Spegni Francesco Spegni is post-doctoral researcher at Universitá Politecnica delle Marche, Italy, where he previously received his PhD degree in Computer Engineering (2011). He received his B.S. and M.S. in Computer Science at Universita‘ degli Studi di Bologna, Italy (2007). He has been visiting fellow at SRI International, California (2010), and TU Wien, Austria (from 2013 till 2015). His research includes model checking of software, as well as parameterized model checking of timed and probabilistic systems.

Luca Spalazzi Luca Spalazzi is associate professor at the Universitá Politecnica delle Marche, Italy. He received the MS degree in electronic engineering (1989) and the PhD degree in artificial intelligent systems (1994) from the University of Ancona, Italy. He worked as a consultant at the IRST-FBK, Trento, Italy, from 1991 to 1993 and in the 1997. He was a visiting scholar at the Australian Artificial Intelligence Institute (AAII), Carlton, Victoria, Australia (1992), and at the Stanford University, California (1996). His research has been supported by grants from the European Union, the Italian Minister of Education, University and Research, the Austrian Research Agency FFG. His present research areas include formal methods and model checking applied to software engineering, cybersecurity and privacy, and multi-agent systems. Regarding the application of formal methods to software engineering, he has worked on the application of semantic model checking to service computing and parameterized model checking to timed systems.