

Semantics-Driven Extraction of Timed Automata from Java Programs

Giovanni Liva · Muhammad Taimoor Khan · Martin Pinzger

Received: date / Accepted: date

Abstract The automatic verification of time properties of models extracted from programs is challenging, mainly because modern programming languages, such as Java, represent time without a proper semantics. Current approaches to extract time models from source code either represent time only as a tree-like sequence of events or require developers to manually provide a formal model of the time behavior. This makes it difficult for software developers to verify various aspects of their systems, such as timeouts, delays and periodicity of the execution. In this paper, we introduce a formal definition of the time semantics for the Java programming language. Based on the semantics, we present an approach to automatically extract timed automata and their time constraints from Java programs at method level. First, our approach detects the Java statements that involve time, from which it then extracts the timed automata. Our extracted automata are directly amenable to the verification of time properties of the corresponding Java methods. We evaluated the accuracy of our approach on twenty open source Java projects that implement time behavior in their source code. The results show that our approach achieves 100% precision and recall in identifying time related information. They also show that 95% of the timed automata extracted from source code correctly model the time behavior of the method. Finally, we show the applicability of our timed automata to identify eight real errors in four open source Apache systems.

Keywords Program Verification · Time Semantics · Timed Automata

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

G. Liva and M. Pinzger
Software Engineering Research Group, Alpen-Adria Universität Klagenfurt, Austria
E-mail: {giovanni.liva, martin.pinzger}@aau.at

M. T. Khan
Surrey Center for CyberSecurity, University of Surrey, UK
E-mail: m.t.khan@surrey.ac.uk

1 Introduction

The quality of software is mainly determined by automatically testing or manually reviewing the software. These activities consume between 40% and 67% of the total project cost depending on the maturity level of the company [37]. Moreover, while these activities help developers to detect the presence of errors they fail to assure the absence of errors in software as said by Dijkstra [20]. A more rigorous way to establish software quality is to *formally* assure that the software is free of critical errors using formal methods. Formal methods require an abstract model of the source code that captures the specific semantics of the implemented functionalities. A particular domain of such a functionality is the time domain. [For instance, developers use time to implement time frames when external events are expected to occur, to set execution timeouts, or to schedule events that occur periodically.](#) A common technique to specify and model the time behavior of programs is the timed automata formalism.

The formal verification of time properties of programs, modeled as timed automata, has been intensively studied over the last 20 years. Timed automata have been introduced by Alur [3] and allow software engineers to model and analyze the timing behavior of programs. For instance, Jayaraman et al. [35] used the timed automata modeled by the developers to monitor the subject system and to verify whether its execution conforms to its specification. Hakimipour et al. [27] present an approach to derive real-time Java programs from the timed automata constructed by the users. Following the same idea, Georgiou et al. [25] presents a technique to translate timed input/output automata into executable Java code. All these research efforts propose techniques that require the developers to provide the formal models and, typically, they are constructed *manually*. Some recent approaches, such as presented in [45, 59], automatically extract such models from Java source code. However, these approaches do not consider the specific semantics of the programming language and fail to correctly model the time domain in programs. For instance, they only retain time as a sequence of events represented in a tree-like structure but fail to model other time aspects of the program, such as timing delays.

In this paper, we address two shortcomings of the aforementioned approaches: (i) they are based on an informal semantics of time and (ii) the timed automata are manually constructed and thus, they are error prone. We propose a technique to automatically extract timed automata from source code by introducing a formal time semantics for the Java programming language. The extracted timed automata are directly amenable to verification aiding developers to verify the correctness of their implementation.

Figure 1 shows an overview of our approach to extract the timed automata. Based on our defined semantics of time, first we parse the source code of a Java method and extract its time related information. Next, we use the extracted time information to build the states and transitions that defines the timed automaton. Our approach works at method level therefore we need a way to model side effects caused by using non-local time variables, such as class attributes. Our approach tackles this problem with dynamic analysis

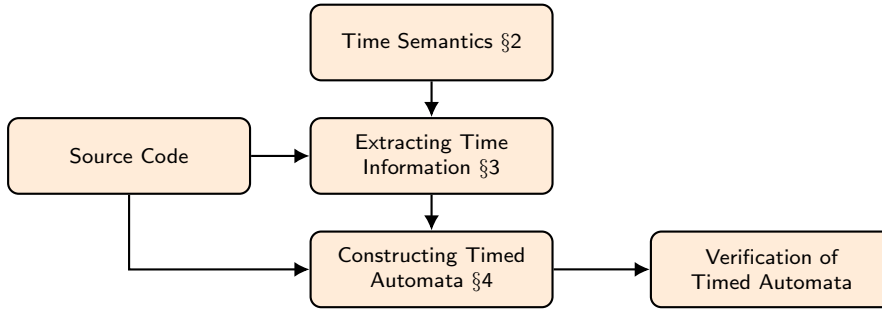


Fig. 1 Overview of our approach to extract a timed automaton of a Java method.

monitoring the values of these statements and expressions during the execution of the test suite. For each run of a test case, our approach generates a set of instances of the timed automaton that can be used for verification.

We have implemented our approach in a prototype tool¹ used to show and study its applicability. We evaluated our approach on 20 open source Java projects. Our experiments demonstrate that the produced timed automata have a precision and recall of 100% in covering time related information. Furthermore, 95% of the timed automata extracted from source code correctly model the time behavior of a method. In addition, the results show that our semantics of time for the Java APIs is sufficient since, in our 20 subject systems, we found that the developers only use the methods provided by the Java APIs to implement time-related behavior. The maximum time for generating a timed automaton took less than 0.5 seconds and had a low memory footprint. Finally, the results show that our timed automata could effectively be used to identify 8 real errors in four open source Java projects.

In summary, this paper makes the following contributions:

- a formal definition of the semantics for the Java 8 (and later) time APIs;
- an approach to automatically extract timed automata for Java methods;
- an evaluation of our approach with 20 open source Java projects;
- an open source implementation of our approach in a prototype tool.

This paper extends our SCAM 2017 paper “Extracting Timed Automata from Java Methods” [43]. In this new version, we have improved the time semantics, addressing several limitations of the original approach and we have extended the previous experiments from 10 to 20 open source Java projects. Moreover, we have added three more research questions (RQ2, RQ3, and RQ5) to further analyze the limitations of our approach. Finally, we have added a description of the methodology to proof the soundness of our translation.

The remainder of the paper is organized as follows: Section 2 presents our time semantics defined for the Java 8 methods. Section 3 presents the application of our semantics to extract time information from Java statements. Section 4 introduces our approach to extract the timed automata from Java

¹ <https://git-isys.aau.at/giliva/java-time-verification>

methods. Section 5 presents the evaluation. The discussion of the results and threats to validity are presented in Section 6. Section 7 presents related work and we conclude the paper in Section 8.

2 Time Semantics

In this section, we first provide background information on the semantics of programming languages. Next, we present our manual categorization of the Java 8 (and later) time APIs. Based on the categorization, we then introduce our time semantics for the Java language.

2.1 Programming Languages and Semantics

The idea of using semantics as a tool for modelling program behaviour is rooted in the work of McCarthy [47] that was used to model the ALGOL [46] and LISP [57] programming languages. Based on this work, Plotkin [51, 52] developed a semantics that capture the notion of a computational step. This approach has been successfully used for formalizing semantics for various programming languages and computational models [1, 17, 24, 31, 48, 54, 60]. Nowadays, all the mainstream languages have their formal semantics, *e.g.*, PHP [23], C [29], C# [16], Python [26], and Javascript [50]. The definition of a formal semantics permits to address different problems, such as the detection of inconsistencies in programs [44], the detection of inconsistencies in the implementation of compilers [58], and monitoring the reliability and security of industrial controllers [36].

Regarding the Java programming language, three different semantics were proposed: ASM-Java [56], JavaFAN [22], and K-Java [14]. ASM-Java was the first complete semantics of Java. It defines a complete semantics for both, the Java language and its bytecode interpretation for the version 1.0 of the programming language. JavaFAN defines only the semantics of the most frequently used features of Java 1.4. The most recent and richest semantics for Java is provided by K-Java presented by Bogdanas and Roşu [14]. They proposed a complete definition of semantics for the Java 1.4 language. In this work, we enrich their definition including the semantics of time for the Java programming language.

All the semantics presented above define the meaning of the structure of a program but they do not provide a meaning for the domain on which the program statements operate. With respect to the work presented in this paper, this domain is *time*. We present a time semantics for the Java programming language that is a rigorous mathematical study of the meaning of the time APIs offered by the language. The semantics describes the processes that the Java Virtual Machine follows when executing the functionality of such APIs. Like many other programming languages, Java provides developers with APIs to implement time related behaviour, such as the Java `java.time.LocalDate`

class. But, Java also provides APIs that allow developers to represent and handle time and timestamps with integer values. The integer representation of time can introduce semantic inconsistencies in a program resulting in failures or preemptive termination. For example, the Java statement `long now = -1`, is syntactically correct and it compiles. However, during the execution of the program, if the variable `now` is used in a timed API call, it will result in a runtime exception. The error is not detected by the Java compiler because it does not understand the semantics of time and that the variable `now` is a time variable since it is used in a timed API call. Therefore, the compiler fails to warn the developer that the variable `now` holds an incorrect time value, namely -1.

As a prerequisite for defining a semantics of time for the Java programming language, we first need to find out which basic methods of Java deal with time, meaning return time or consume time as a parameter. This is presented in the next subsection.

2.2 Classification of Time Related Java 8 Methods

For defining the time semantics, we manually analyzed the API documentation of Java 8 classes and identified all API methods that either return time as an integer value or consume time as an integer value in one of their parameters. We also performed this analysis with the API documentation of the Java versions 9 and 10 and we discovered that the time APIs did not change. Therefore, our semantics support also the most recent versions of Java. Next, we classified the found methods into the following four categories:

1. **Return Time (RT):** The first category covers methods that return an integer value that represents time. For example, the static `currentTimeMillis()` method of the `System` class returns the current time in milliseconds.
2. **Explicit Time (ET):** The second category covers methods that contain a time parameter declared as integer. For example, in the `connect` method of the `Socket` class, if called with two parameters, the second parameter specifies the maximum amount of time for establishing the connection.
3. **Explicit Wait (EW):** The third category is composed of methods that wait for an external event to continue the execution of a thread. Since it is possible that this event might never occur, these methods can potentially block the execution of a thread forever. Examples are the `wait()` method of the `Object` class or the method `get` of the `Future` class.
4. **Set Timeout (ST):** The fourth category comprises methods which change their behavior with a time constraint that is set by a preceding method call. For example, the `connection` method of the `URLConnection` class by default has no upper time limit for establishing the connection. Instead, a timeout can be set by calling the `setConnectTimeout` method before calling the `connection` method.

Table 1 Excerpt of time related methods found in the Java 8 APIs.

Return Time (RT)		Explicit Time (ET)	
Class	Method Signature	Class	Method Signature
System	nanoTime()	Socket	connect(SocketAddress,int)
System	currentTimeMillis()	Thread	join(long)
Clock	millis()	Thread	join(long,int)
Duration	get(TemporalUnit)	Thread	sleep(long)
Duration	getNano()	Thread	sleep(long,int)
Duration	getSeconds()	FutureTask	get(long,TimeUnit)
...

Explicit Wait (EW)		Set Timeout (ST)	
Class	Method Signature	Class	Method Signature
Object	wait()	InputStream	read()
Future	get()	InputStream	read(byte[])
FutureTask	get()	InputStream	read(byte[],int,int)
Thread	join()	URLConnection	connect()
Process	waitFor()	ServerSocket	accept()
CountDownLatch	await()	SSLServerSocket	accept()
...

Table 1 presents an excerpt² of the analyzed Java 8 APIs methods that involve time. In this paper, we focus on the first three categories of time related methods and we provide a semantics for them. For the category **ST**, it is not always possible, using static analysis, to correctly detect whether a method call has a specific timeout set by a preceding method call. Therefore, we skip this category of methods for this work and will address them in our future work.

2.3 Semantics of Java Time Statements

Based on the classification of time related methods of the Java 8 APIs, we introduce the semantics of time for the statements in Java programs. We start with introducing the general concepts for representing time variables and time related methods.

A *time variable* is a program variable, which stores time values. We model the time as positive natural number and we define the set of time variables as V^t , such that $\forall v \in V^t. value(v) \in \mathbb{N}^+$, where $value(\cdot)$ refers to the value held by the variable v . We define a set of Java method definitions with the letter M where the superscript t is used to define that they are time related and the subscript is used to define the category to which a method belongs: M_{rt}^t denotes the set of **RT** methods that return time; M_{et}^t denotes the set of **ET** methods that have a time parameter in their signature; and M_{ew}^t denotes the set of **EW** methods that can potentially block the execution of a thread

² The full list is available at <https://git.io/fAbZR>

forever. Finally, we call the quadruplet V^t, M_{rt}^t, M_{et}^t , and M_{ew}^t an *environment* and we denote it with the letter E . The environment keeps track of the time information extracted from the source code.

$$E = \langle V^t, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle$$

We define the time semantics as a set of operational semantics [52] rules. A rule has some premises that constitute the preconditions to apply the rule. The premises have an environment E and the program statement S . Based on this, we can apply the rules R_1, \dots, R_n that can conclude a new environment E' .

$$\frac{R_1(E, S), \dots, R_n(E, S) \vdash \langle E' \rangle}{\langle E, S \rangle \vdash \langle E' \rangle}$$

Given the time domain \mathbb{T} defined as positive integer numbers and the current point in time $t_0 \in \mathbb{T}$, we define the rules T_1, T_2 , and T_3 to model the time semantics of the three categories of methods considered by our approach. These three rules describe how the time changes given the couple (V^t, t_0) , that represent the list of time variables and the current point in time, and the API method call. We define $val(\cdot)$ as the function which returns the value of the input expression that can be either a reference to a variable or a method call. The rule T_1 handles the assignment of time returned by calls to methods of the category **RT** updating the value stored by the time variable x without touching the execution time. It is defined as:

$$T_1: \frac{x \in V^t \quad m \in M_{rt}^t}{\langle (V^t, t_0), x = m(_) \rangle \vdash \langle V^t[x \setminus val(m(_))], t_0 \rangle}$$

Rule T_2 handles **ET** method calls containing a time argument represented by the variable t . At the end of the execution of the method m , the time is increased by value t :

$$T_2: \frac{m \in M_{et}^t}{\langle (V^t, t_0), x = m(_, t, _) \rangle \vdash \langle V^t, t_0 + val(t) \rangle}$$

Finally, rule T_3 handles calls to **EW** methods that could potentially block the execution of a thread forever. It is defined as:

$$T_3: \frac{m \in M_{ew}^t}{\langle (V^t, t_0), x = m(_, t, _) \rangle \vdash \langle V^t, \infty \rangle}$$

In the next section, we present how our time semantics is used to infer time related Java methods, statements, and time variables.

```

1: public class Cache {
2:     int value;
3:     long lastRefresh;
4:     static final int MAX_TIME = 10*60*1000; //10minutes
5:     public Cache(){
6:         value = ExternalClass.readValue();
7:         lastRefresh = now();
8:     }
9:     public int read(){
10:        long now = System.currentTimeMillis();
11:        if(now - lastRefresh > MAX_TIME){
12:            value = ExternalClass.readValue();
13:            lastRefresh = now();
14:        }
15:        return value;
16:    }
17:    public long now() {
18:        return System.currentTimeMillis();
19:    }
20: }

```

Fig. 2 Code example that uses the Java time API.

3 Extracting Time Information

Based on the time semantics presented in the previous section, we define a set of rules to analyze the source code of a project. Through the analysis, our approach gathers the methods and statements that are time related and program variables that are time variables. We group the rules into three categories:

- time methods analysis,
- branching statements analysis,
- expressions analysis.

Note, these three categories of rules are applied to the source code until no more time methods, statements, or time variables are found. Vice versa, when new time methods, statements, or variables are found, then the additional information is used to perform another analysis round.

Figure 2 shows an example of using the time API in Java. The class `Cache` implements a simple cache that stores the value returned by the invocation of `ExternalClass.readValue()`. When the method `read()` is called, the code verifies how much time elapsed since the last call to `readValue()`. If the elapsed time is greater than a threshold `MAX_TIME`, the cached value is refreshed. We use this example to show how our semantics is applied to determine time methods, statements, and variables.

3.1 Time Methods Analysis

Using the definitions given in Section 2.3, we first introduce a set of rules to gather the methods of the categories **RT** and **ET** in the source code of a Java

program. Methods that return time are added to M_{rt}^t . Methods that contain a time variable as a parameter are added to M_{et}^t . Note, since we did not provide a formal semantics for **ST** methods, our approach does not consider this category of methods. Furthermore, while our approach supports the Java APIs **EW** methods, it does currently not cover **EW** methods implemented by developers. For gathering the developer's defined **RT** and **ET** methods, our approach applies the rules R_{rt} and R_{et} detailed in the following.

Rule R_{rt} : This rule matches the project's **RT** methods that return time and adds them to the set M_{rt}^t . It is defined as:

$$R_{rt}: \frac{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), S \rangle \vdash \langle V^{t'}, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle \quad \exists r : \text{return}(S).(\text{isVar}(r) \wedge r \in V^{t'}) \vee (\text{isCall}(r) \wedge r \in M_{rt}^t)}{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), \text{name}(\text{pars}) \ S \rangle \vdash \langle V^t, \{\text{name}\} \cup M_{rt}^t, M_{et}^t, M_{ew}^t \rangle}$$

Given a method called *name* with the list of parameters *pars* and the method body *S*, our approach applies the time semantics recursively to the body *S*. If *S* contains a return statement that references a time variable or a call to an **RT** method, the method *name* is added to the set M_{rt}^t . The function *isVar*(\cdot) returns true if the expression is a reference to a variable. Similarly, *isCall*(\cdot) returns true if the expression is a method call. The function *return*(\cdot) obtains the return statements of the given method body *S*.

Example: Considering the method **now** defined at Line 17 of Figure 2, the rule R_{rt} is matched as follows:

$$R_{rt}: \frac{\langle (_, M_{rt}^t, _, _), \{ \text{return Sys.currTime();} \} \rangle \vdash \langle _, M_{rt}^t, _, _ \rangle \quad r := \text{Sys.currTime().}(_ \vee (\text{isCall}(r) \wedge r \in M_{rt}^t))}{\langle (_, M_{rt}^t, _, _), \text{now}() \{ \text{return Sys.currTime();} \} \rangle \vdash \langle _, \{\text{now}\} \cup M_{rt}^t, _, _ \rangle}$$

The premise of the rule R_{rt} matches the variable *name* with the method name **now**, the list of parameter *pars* with an empty list, and the body *S* with the return statement of the method. For readability, we have shortened the method call to **System.currentTimeMillis()** in the return statement. The rule analyzes the body of the method marking the method call in the return statement as an **RT** method call. When the rule retrieves the expression of the return statement in the variable *r*, it validates the second branch of the *or* clause satisfying all the judgments of the rule. Therefore, the fully qualified name of method **now** is added to the list of **RT** methods.

Rule R_{et} : This rule identifies the project's **ET** methods that accept a time parameter and adds them to the set M_{et}^t . It is defined as:

$$R_{et}: \frac{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), S \rangle \vdash \langle V^{t'}, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle \quad \exists p : \text{pars}.p \in V^{t'}}{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), \text{name}(\text{pars}) \ S \rangle \vdash \langle V^{t'}, M_{rt}^t, \{\text{name}\} \cup M_{et}^t, M_{ew}^t \rangle}$$

Given a method called *name* with the list of parameters *pars* and the method body *S*, our approach adds it to M_{et}^t if the set of time variables V^t of the environment, resulting from the application of our time semantics to the body *S*, contains a parameter declared by method *name*.

Both rules, R_{rt} and R_{et} , require the application of our time semantics to the statements *S* of the method body to determine whether a method is of category **RT** or **ET**. Our approach applies the time semantics to the statements following the order of the method's control flow. Java provides different types of statements, such as conditional and loop statements, lambda expressions, ternary operator, assignment and mathematical expressions. In the next subsections, we provide the rules to handle the relevant Java statements. The other types of statements are handled by our approach in a similar way therefore we omit the rules for them.

3.2 Branching Statements Analysis

In this subsection, we present the relevant rules for the two branching statements *if* and *while*. The other branching statements of Java, such as *try-catch* and *do-while*, follow the same approach.

Rules R_{if} and R_{loop} : The idea of the rules R_{if} and R_{loop} is to first apply the time semantics to the guard *B* and then to the body *S*, collecting the time information. Our approach applies these rules until a fix point is reached and no more time information is collected in the environment. For the *if* branching statement, the time information collected from both branches is unified.

$$R_{if}: \frac{\langle E, B \rangle \vdash \langle E_0 \rangle \quad \langle E_0, S' \rangle \vdash \langle E' \rangle \quad \langle E_0, S'' \rangle \vdash \langle E'' \rangle}{\langle E, (\mathbf{if} (B) \{ S' \} \mathbf{else} \{ S'' \}) \rangle \vdash \langle E' \cup E'' \rangle}$$

$$R_{loop}: \frac{\langle E, B \rangle \vdash \langle E' \rangle \quad \langle E', S \rangle \vdash \langle E'' \rangle}{\langle E, (\mathbf{while} (B) \{ S \}) \rangle \vdash \langle E'' \rangle}$$

Example: Considering the if statement in Lines 11–14 of Figure 2, the rule R_{if} matches as follows:

$$R_{if}: \frac{\langle E_0, S \rangle \vdash \langle E' \rangle \quad \langle E_0, \emptyset \rangle \vdash \langle E_0 \rangle}{\langle E, \mathbf{now-lastRefresh} > \mathbf{MAX_TIME} \rangle \vdash \langle E_0 \rangle}$$

$$\frac{\langle E, \mathbf{now-lastRefresh} > \mathbf{MAX_TIME} \rangle \vdash \langle E_0 \rangle}{\langle E, (\mathbf{if} (\mathbf{now-lastRefresh} > \mathbf{MAX_TIME}) \{ S \} \mathbf{else} \{ \}) \rangle \vdash \langle E' \cup E_0 \rangle}$$

The guard *B* matches the expression `now - lastRefresh > MAX_TIME`, the then-body *S'* matches the list of statements in Lines 12–13, and the else-body *S''* matches an empty list of statements. For readability, we omit the statements in the then-body. The rule first analyzes the guard expression recursively using the rules presented in the next subsection. The result is an environment E' with the time variables referenced in the guard expression. Then, the rule uses the environment E as input to analyze the then- and else-body recursively. Since the else-body is empty, the resulting environment does not change.

3.3 Expressions Analysis

The following presents the set of rules to analyze Java expressions to identify time related variables. First, we present how to process boolean expressions and then how to process assignment and mathematical expressions.

In a boolean expression, there are two different cases in which our approach marks a program variable as time variable: (i) the variable is compared with a time variable or (ii) the variable is compared with the result of an **RT** method call. If an expression consists of multiple boolean expressions, we process each expression separately. Considering the **if** statement at Line 11 of Figure 2, given that variable **now** defined in Line 10 is a time variable, our semantics analyzes first the boolean guard and then the then-branch. In the guard, the variable **now** is compared with the constant **MAX.TIME** and thus, our semantics marks the constant **MAX.TIME** as time related variable.

Regarding assignment and mathematical expressions, there are three different cases in which a programming variable is used as time variable:

- The variable is assigned the result of an **RT** method. This case is handled by the rule R_1 .
- The variable is used as timeout parameter in an **ET** method call. This case is handled by the rule R_2 .
- The variable is used in a mathematical expression with other time variables or time methods. This case is handled by the rule R_3 .

Rule R_1 : The rule R_1 verifies that the return value of method m is assigned to the variable x and m is a method of the **RT** category. If this condition holds, x is added to the set of time variables V^t .

$$R_1: \frac{m \in M_{rt}^t}{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), x = m(_) \rangle \vdash \langle \{x\} \cup V^t, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle}$$

Example: Line 10 in Figure 2 shows a statement on which this rule applies. Variable **now** is assigned the result of the call to the **RT** method **System.currentTimeMillis()** and therefore, our semantics adds **now** as time variable.

$$R_1: \frac{\text{Sys.currTime}() \in M_{rt}^t}{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), x = \text{Sys.currTime}(); \rangle \vdash \langle \{x\} \cup V^t, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle}$$

Rule R_2 : For the second case, we define two auxiliary functions to simplify the readability of our semantics. Function $pos(\cdot)$ returns the position of the input parameter in the method call and function $timeoutpars(\cdot)$ returns the set of indices of the time parameters of the input method. The rule marks as time variable every variable that is passed as a time parameter in a method call of the category **ET**.

$$R_2: \frac{m \in M_{et}^t \wedge pos(y) \in timeoutpars(m)}{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), m(_, y, _) \rangle \vdash \langle \{y\} \cup V^t, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle}$$

Rule R_3 : The last case requires to consider different types of operands of mathematical expressions:

1. Both operands are time variables.
2. One operand is a time variable and the other is a normal variable.
3. One operand is a time variable and the other is a scalar value.
4. One operand is a normal variable and the other is an **RT** method call.
5. Both operands are **RT** method calls.
6. One operand is an **RT** method call and the other is a scalar value.

Each case is encoded in one of the following rules, where op_i is an operand of the mathematical expression:

$$\begin{aligned}
 e_1 &= op_1 \in V^t \wedge op_2 \in V^t \\
 e_2 &= (op_1 \in V^t \vee op_2 \in V^t) \wedge (isVar(op_1) \vee isVar(op_2)) \\
 e_3 &= (op_1 \in V^t \wedge op_2 \in \mathbb{N}^+) \vee (op_1 \in \mathbb{N}^+ \wedge op_2 \in V^t) \\
 e_4 &= (isVar(op_1) \wedge op_2 \in M_{rt}^t) \vee (op_1 \in M_{rt}^t \wedge isVar(op_2)) \\
 e_5 &= op_1 \in M_{rt}^t \wedge op_2 \in M_{rt}^t \\
 e_6 &= (op_1 \in M_{rt}^t \wedge op_2 \in \mathbb{N}^+) \vee (op_1 \in \mathbb{N}^+ \wedge op_2 \in M_{rt}^t)
 \end{aligned}$$

If any of the previous rules holds, then the variable used in the assignment statement is added to the set of time variables. Moreover, the rule R_3 checks that the return type of the expression is numeric because Java allows using the "+" operator with other data types, *e.g.*, it can be used to concatenate strings. This control is necessary to avoid adding variables that do not hold time values.

$$R_3: \frac{e_1 \vee e_2 \vee e_3 \vee e_4 \vee e_5 \vee e_6 \quad \odot \in \{+, -, /, *\} \quad is_numeric(op_1 \odot op_2)}{\langle (V^t, M_{rt}^t, M_{et}^t, M_{ew}^t), x = op_1 \odot op_2 \rangle \vdash \langle \{x, op_1, op_2\} \cup V^t, M_{rt}^t, M_{et}^t, M_{ew}^t \rangle}$$

Example: The guard expression in Line 11 of Figure 2 shows an instance where this rule applies. The left-hand-side of the condition contains the mathematical expression `now - lastRefresh`. Using rule R_1 in Line 10, the variable `now` is added to the set of time variables V^t . When the guard expression is processed with the rule R_3 , expression e_2 is satisfied and the return value is numeric. Therefore, the rule adds the variable `lastRefresh` to the set of time variables.

$$R_3: \frac{(\text{now} \in V^t \wedge isVar(\text{lastRefresh})) \vee _ \quad is_numeric(\text{now} - \text{lastRefresh})}{\langle (V^t, _, _, _), \text{now} - \text{lastRefresh} \rangle \vdash \langle \{\text{now}, \text{lastRefresh}\} \cup V^t, _, _, _ \rangle}$$

4 Constructing Timed Automata

With the time semantics presented in the previous section, our approach is able to extract the time information necessary to generate a timed automaton for a Java method. There are multiple temporal logics available that can be used to describe properties to be verified on the extracted timed automaton. We have decided to use Timed Computation Tree Logic (TCTL) [5, 8] because it provides means to describe properties related to time and the execution state and path. Currently, there exist two model checkers for timed automata that support TCTL: Kronos [62] and UPPAAL [39]. We decided to use UPPAAL because it has more than 15 years of development and runs on Java 8, while the last release of Kronos dates back to 2002 and we were not able to execute it on our machines. Furthermore, UPPAAL has been successfully used in several case studies, for instance to prove the reliability of gearbox controllers [42], to correctly synthesize control programs for batch production [34], and to verify the correctness of audio protocols [38, 9].

UPPAAL extends the classical timed automaton theory defined by Alur and Dill [4] with syntactic sugar, *e.g.*, committed location, and additional features, *e.g.*, integer variables and channels. UPPAAL uses a transition system in which a transition can have three properties: (i) a time constraint, (ii) a reset of time variables, and (iii) an update of variables. Time variables in the context of timed automata are called *clock* variables. A time constraint specifies an extra condition involving clock variables. A transition is enabled only if the time constraint is satisfied. The reset option specifies the list of clock variables to reset their values to zero. The update option assigns values to variables. The reset and update actions are performed only when the transition is fired. Moreover, a timed automaton has a set of clock variables that model the flow of time. In our approach, we use a single clock variable t_0 that keeps track of the execution time.

Our approach to extract UPPAAL timed automata consists of the following three steps:

1. construct the initial timed automaton with states and transitions
2. refine the automaton with additional time constraints; and
3. finalize the automaton.

In the following subsections, we describe each step of our approach in detail.

4.1 Construct the Initial Timed Automaton

The construction of the initial timed automaton starts with generating states and transitions following a standard procedure based on the control-flow graph (CFG) of the method's source code.

For each statement node in the CFG, our approach creates a respective *committed* state in the timed automaton. A *committed* state freezes the time simulating that the execution of the code is instantaneous. [This prevents the](#)

interleaving of actions that require atomicity. However, if the CFG node contains a statement with an **ET**/**EW** method call, the state created in the timed automaton is a normal state. This models the fact that in this state the time is important and it can elapse. Using committed states helps the model checker to perform internal optimizations and it reduces significantly the state space that it has to explore. Furthermore, we mark the root node of the control-flow graph as initial state of the automaton.

Similarly, for each connection between two nodes in the CFG, our approach creates a transition that connects the respective states in the automaton. All the transitions extracted in this step are ϵ -transitions without any constraints. The decision of using ϵ -transitions can introduce a problem with branching instructions and resulting in a non-deterministic automaton. Since we are interested only in time properties of the code, which path is taken in an automaton run is not important. However, as presented in detail later, if a branching instruction in the code is guarded by a time expression, our approach adds the time expression as constraint to the transition of the then-branch. Moreover, for each transition from state s to s' , our approach verifies if state s contains an expression that assigns a value to a time variable as defined by rules R_1 and R_3 . If this is the case, the assignment is added to the transition as update property.

4.2 Refine the Automaton with Time Constraints

After constructing an initial timed automaton for a Java method, our approach analyzes the time statements in the CFG to infer time constraints that are used to refine the transitions of the initial automaton. Currently, our approach infers the following three types of time constraints:

- Timeout Constraint: covers calls to **ET** methods that contain an argument defining the maximum waiting time, such as `Thread.sleep(2000)`.
- Indefinite Wait Constraint: covers calls to **EW** methods that can stop the execution of a method forever, such as `Thread.join()`.
- Time Expired Constraint: covers conditional statements that trigger the execution of a specific piece of code, such as `if (ticks < MAX_TICKS) { ... }`.

Based on the inferred time constraints, the approach updates the corresponding transition in the initial automaton from state s to state s' accordingly.

Timeout Constraint: Method calls to **ET** methods contain an argument that specifies the maximal waiting time for the termination of the method call. This represents a Timeout Constraint that sets the maximum execution time for that method call. For each statement in the CFG that calls a method of the category **ET** our semantics is used to first extract the timeout parameters. Next, for each timeout parameter *timeout* our approach adds a time constraint

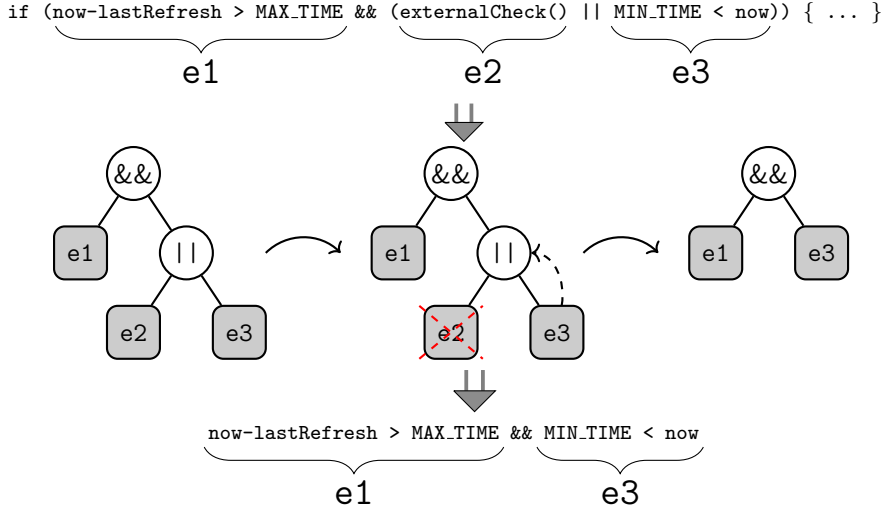


Fig. 3 Example of extracting the time related expression from the if-condition in which `e1` and `e3` are time related and `e2` is not.

$t_0 \leq \text{timeout}$ to all the transitions exiting from state s that represents that method call in the automaton. Moreover, all the transitions that are entering in that state s are modified inserting the reset of variable t_0 . With this model, when the automaton run enters state s , the clock variable that keeps track of the execution time will block the execution of the automaton for a maximum of *timeout* time.

Indefinite Wait Constraint: Method calls to **EW** methods could possibly block the execution of a method forever. If our approach identifies an **EW** method call in the CFG, it adds an ϵ -transition from state s to s that represents that method call in the automaton. The self loop models the fact that the automaton can either proceed to the next state or stay forever in the same state.

Time Expired Constraint: The condition of a branching instruction can contain a time constraint to decide which branch to execute. An example of such a time constraint is given by the if-condition at Line 11 of Figure 2. If enough time has passed, it executes the code in the then-branch. Our approach models this using the following algorithm: First, it parses the condition and constructs the Abstract Syntax Tree (AST) of the time expression as depicted by the first tree on the left hand-side of Figure 3. Next, it applies our time semantics to identify expressions that are time related (*i.e.*, contain a time variable or a call to an **RT** method that returns time). In this example, the expressions `e1` and `e3` are time related. All the nodes that do not contain a time expression are removed from the tree. In our example, the node `e2` is removed. Next, the algorithm removes the nodes representing the boolean operators from which one or both child nodes have been removed in the previous step pushing up

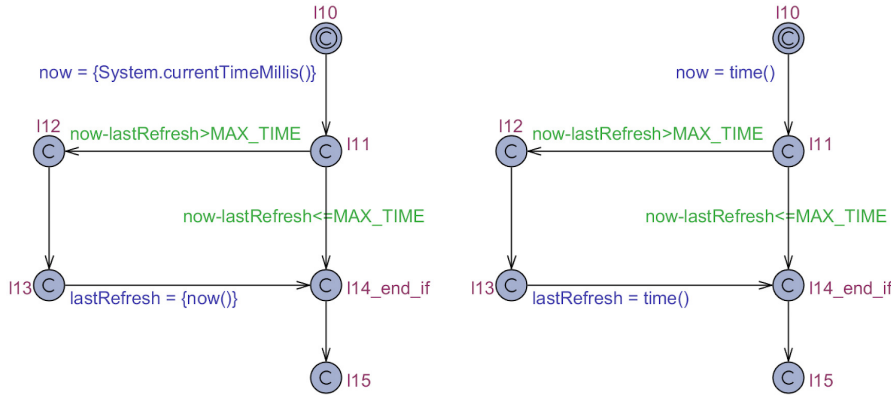


Fig. 4 The graph on the left depicts the timed automaton extracted from the source code of the `read()` method presented in Figure 2. State names represent the source code line numbers. Expressions that are rewritten or replaced are contained between curly brackets. The graph on the right depicts an instance of this automaton after rewriting/replacing the marked expressions.

the remaining child, if present. In the example shown on the right-hand side of Figure 3, the node "||" is removed because its expression `e2` has been removed and `e3` has been pushed up. Finally, the resulting tree is pretty-printed as a string that it is added as a time constraint to the transition representing the then-branch. Furthermore, the negated version of it is added as a time constraint to the transition representing the else-branch. This way our approach creates a deterministic automaton for branching instructions that are time related.

Finally, we translate the refined timed automaton into the corresponding UPPAAL model. However, the translated model may not be amenable to the verification because it may include some expressions which are not directly supported by UPPAAL. We address this issue in the next sub-section.

4.3 Finalize the Timed Automaton

The final step of our approach is to rewrite the expressions that have been copied from the source code as correct UPPAAL expressions. Our approach finalizes the previously constructed UPPAAL timed automaton by formalizing those time expressions, which are not supported by UPPAAL, *i.e.*, (i) time variables and (ii) **RT** method calls. They need to be replaced by concrete values or UPPAAL expressions.

The graph on the left hand side of Figure 4 shows an example of this problem with the automaton extracted from the source code of the method `read()` displayed in Figure 2. The calls to `System.currentTimeMillis()` and `now()` are surrounded by curly brackets to easily identify and later replace them. UPPAAL has no definition of these two Java methods and the three

variables `now`, `lastRefresh`, and `MAX.TIME`. Therefore, we need to translate them into expressions that are processable by UPPAAL, whose details are discussed in the following subsections.

Time variables: In our approach, we model time variables defining them as UPPAAL variables using its integer theory. For time variables that are declared and initialized in the Java source code, our approach copies the Java initialization expression as initialization for the UPPAAL variables. Otherwise the UPPAAL variables are initialized with the value 0. This however does not block our approach to correctly model the behavior of program variables that can be modified outside of the scope of the method. For instance, the variable `lastRefresh` is an instance variable that can be changed by other methods. We refer to these time variables as non-local time variables. They are handled by our approach separately, as we describe below.

RT method calls: This category of method calls cannot be directly translated because UPPAAL does not come with a definition for those methods. For instance, it does not have a definition for the method `System.currentTimeMillis()`. In our manual analysis of Java 8 APIs, we identified which of Java’s **RT** methods return a monotonic time value. For these methods, we provide the UPPAAL defined function *time* that simulates the same behavior – every time it is called it returns a monotonically increasing natural number. Using this definition, our approach traverses the automaton and replaces all calls to such **RT** methods within curly brackets with a call to the *time* function.

Non-local time variables and unresolved method calls: For the other method calls and non-local time variables contained in our automaton, our approach currently does not provide any formal model in UPPAAL. It addresses this limitation with dynamic analysis to monitor the program during its execution and to obtain representative values for these expressions. Then, our approach replaces these method calls and the initialization values of non-local time variables with the monitored values creating an automaton that can be model checked with UPPAAL.

Our approach performs the following steps: First, during the extraction of the automaton, our approach records the expressions that it could not translate into UPPAAL expressions. For each expression, our approach records the line number in the source file, the fully qualified names of the class and method containing that expression, and the expression itself. We implemented this method as an agent that is added to the Java Virtual Machine.

At the initialization phase of the Java Virtual Machine, the agent loads the list of recorded expressions and rewrites the bytecode of the classes that contain these expressions inserting logging statements. For the non-local time variables, a logging statement is inserted at the beginning of the method recording the value of each non-local time variable when the method is called. For each unresolved method call, it inserts the logging statement at the line where the method is called. Each logging statement outputs the thread id, fully qualified

names of the class and method, line number in the source code, the expression monitored, and the time value of the non-local time variable or method call to a log file. For each execution of a program such a log file is created.

Finally, for each thread and for each method execution, it creates an instance of the respective timed automaton replacing the non-local time variables and method calls with their recorded values.

The graph on the right hand side of Figure 4 depicts an instance of a timed automaton created for the `read()` method in Figure 2 with the correct syntax and semantic for the variables `now` and `lastRefresh`, and the `System.currentTimeMillis()` and `now()` method calls. This automaton can be used with UPPAAL to formally verify the time properties of this method, such as termination.

5 Evaluation

In this section, we present the experiments we have performed to evaluate our approach to extract timed automata from the source code of Java methods. With the results we aim to answer the following five research questions:

- **RQ1:** What is the precision and recall of our approach to extract time information from source code?
- **RQ2:** What is the ratio of methods whose time behaviour depends on non-local time variables?
- **RQ3:** What is the number of method calls to time methods in external libraries?
- **RQ4:** Are the extracted models adequate to detect real errors in open source projects?
- **RQ5:** How much time and memory is required by our approach to extract the timed automata?

The first research question is used to evaluate the precision and recall of our approach to extract time information from source code. With respect to our time semantic defined for the Java 8 API, we expect our approach to extract this information with high precision and recall. The second and third research questions are used to evaluate the impact of two limitations of our approach, namely the usages of non-local time variables and external time libraries to implement time related functionality. The fourth research question investigates the capability of our approach to detect "real" errors in Java methods. Finally, the last research question studies how different metrics impact the time and memory needed by our approach to extract a timed automaton from a Java method. In the following, we first describe the dataset used in our experiments and then present the specific set-up for each research question and the results.

Table 2 Open source Java projects used in our evaluation with number of Java files, number of classes (NC), number of methods (NM), and number of methods containing a call to a Java 8 SDK method implementing time (NMT).

Project	Version	Files	NC	NM	NMT
ActiveMQ	5.16.0-SNAPSHOT	4,434	5,071	43,520	1,480
Activiti	7.0.0-SNAPSHOT	2,002	2,103	15,373	66
Airavata	0.17-SNAPSHOT	1,621	9,320	70,862	154
Alluxio	1.6.0-SNAPSHOT	1,319	3,364	24,972	230
Atmosphere	3.0.0-SNAPSHOT	348	500	4,100	54
AWS-SDK-Java	1.11.165-SNAPSHOT	26,416	27,208	205,413	157
Beam	2.2.0-SNAPSHOT	1,696	3,844	21,401	85
Camel	2.20.0-SNAPSHOT	17,205	20,024	116,032	1,277
Elastic-Job	2.1.5	571	611	2,497	24
Flume	1.8.0-SNAPSHOT	642	995	6,705	338
Hadoop	3.0.0-beta1-SNAPSHOT	8,061	12,597	100,589	2,686
Hazelcast	3.9-SNAPSHOT	5,696	7,663	59,260	995
HBase	3.0.0-SNAPSHOT	3,638	9,535	128,914	1,500
Jetty	9.4.7-SNAPSHOT	2,567	3,781	25,548	980
Kafka	0.11.1.0-SNAPSHOT	1,315	1,896	14,006	326
Lens	2.8.0-SNAPSHOT	845	1,036	8,110	177
NanoHTTPD	2.3.2-SNAPSHOT	87	124	716	17
Neo4j	3.3.0-SNAPSHOT	6,681	9,158	61,393	655
Sling	9-SNAPSHOT	5,336	5,964	37,415	530
Twitter4j	4.0.6	426	418	4,641	41
Total		90,906	125,212	951,467	11,772

5.1 Project Selection

For our experiments, we used a set of 20 open source popular Java projects that have been also used in other time related studies [44].

ActiveMQ is a message broker and Activiti is a light-weight workflow and business process management platform. Airavata is a software suite to compose, manage, execute, and monitor large scale applications and workflows on computational resources. Alluxio enables any application to interact with any data from any storage system at memory speed. Atmosphere is a framework to develop client and server side components for building Asynchronous Web Applications and AWS-SDK-Java provides APIs to interact with many Amazon web services. Beam is a unified model for defining data-parallel processing pipelines and Camel is a framework to implement routing and mediation rules in Java- or Scala-based domain specific language. Elastic-Job is a project for running distributed scheduled jobs. Flume is a distributed service for collecting and aggregating log data. Hadoop is a map-reduce implementation and on top of its distributed file system, HBase builds a distributed database. Hazelcast is a clustering and highly scalable data distribution platform. Jetty is a web server provided by the Eclipse Foundation. Kafka provides a unified layer for handling real-time data feeds and, similarly, Lens provides a unified analytics interface from different data sources. Nanohttpd is a light-weight HTTP server designed for embedding in other applications and Neo4j is a graph database. Sling is a web framework that uses a Java Content Repository to store and manage content. Finally, Twitter4j is a Twitter API binding library for the

Java language. These projects use the Java time APIs for scheduling the communication between different components of a system, synchronize activities among different instances of a program, wait for an event to arrive, and handling communication failures over the network.

As can be seen by the descriptive statistics presented in Table 2, the size of the projects varies from 124 to 27,208 classes (column NC), whereas AWS-SDK-Java is the largest project. Only a small fraction of the methods implemented in each project contain a call to a Java 8 time method as indicated by the numbers in column NMT. In total, our semantics identified 11,772 time methods which yields a ratio of 1.24%. The project with the largest number of methods that contain a call to a Java 8 time method is Hadoop with 2,686 methods. The project with the largest percentage of such methods is Flume with roughly 5% of the methods or 338 out of 6,705 methods. The set of 11,772 methods represent the basic input to our experiments.

5.2 RQ1 - What is the precision and recall of our approach to extract time information from source code?

With this research question, we seek to evaluate the precision and recall of our approach in extracting time information from Java source code. The time information that can be extracted from source code are time constraints and time assignments that are then added to the timed automaton.

To evaluate the precision and recall of the generated timed automata, we follow the evaluation methodology presented by Tien-Duy *et al.* [40]. First, the authors of the paper and an independent developer, who has several year of academic and professional experience in developing Java applications, create a reference set of timed automata. The reference set was created with a manual control- and data-flow analysis of 400 methods randomly selected from the 11,772 time methods. The manual data- and control-flow analysis has been performed independently by each researcher and the developer. The results and in particular the discrepancies were discussed in a follow-up meeting. For each discrepancy, the participants together analyzed the corresponding source code again until a consensus was reached. The resulting timed automata represent our *ground truth* models.

Precision and recall then were computed by comparing each timed automaton extracted by our approach with the respective ground truth model. Precision refers to the portion of time constraints and time assignments that were present in the extracted timed automaton that were also present in the ground truth model. Recall refers to the portion of time constraints and time assignments that were present in the ground truth model that were also present in the extracted timed automaton. The result of this analysis shows a perfect precision and recall of 100%.

5.3 RQ2 - What is the ratio of methods whose time behaviour depends on non-local time variables?

Our approach extracts timed automata at method level but it does not consider time information stored in non-local time variables except for time constants. Based on our manual observations of the various methods in the 20 Java projects, we conjecture that developers tend to use time variables locally, except for time constants. The goal of this experiment is to verify our assumption by investigating to which extent developers use time variables locally, *i.e.*, within methods, and to which extent non-local time variables are used and changed. An example of such a non-local time variable is given by the class attribute `lastRefresh` of class `Cache` in Figure 2. In other words, we seek to investigate the ratio of pure methods w.r.t. time variables. Pure methods are methods that have the following two characteristics:

1. The result of a method only depends on the values of its parameters.
2. The execution of the method will not alter the value of any variable defined outside the scope of the method.

A pure method w.r.t. time variables has multiple benefits: they are secure, idempotent, easier to reason about, and easier to test, *i.e.*, it is not necessary to set the system in a specific state since the result depends only on the input parameters.

For this experiment, we employ Daikon [21], an invariant detection tool. Daikon has been used in many previous research efforts, such as in [11, 13, 2, 10, 12, 41, 55, 6], to extract likely program invariants. It runs a program and observes the values that the program computes. During the observation, it applies some logic theories to infer which properties, *i.e.*, invariants, are true over the observed execution. One of the many invariants extracted by Daikon concerns the values of variables. It detects if a variable never changes its value in a method during its execution. We can exploit this invariant to detect which non-local time variables are not modified in the execution of a method.

We randomly selected 400 methods from 11,242 time methods of the projects presented in Table 2. We excluded the 530 methods of Apache Sling because this project requires an external library that is also used by Daikon. Unfortunately, Apache Sling and Daikon depend on different versions of this library causing the tests of Sling to be not executable. Furthermore note, the list of 400 methods used for this research question is different from the list used in Section 5.2.

For each of the 400 methods, we ran the test suite of the respective project with Daikon and stored the invariants that it extracted. Next, we parsed the extracted invariants to check whether the method under study does alter the value of any non-local time variable. The results of this study show that only 19 out of 400 (4.75%) methods alter the value of a non-local time variable while 95.25% (381/400) of the methods alter the value of a local time variable. This answers our research question RQ2: only 4.75% of the time methods alter the value of a non-local time variable during its execution. This low ratio confirms

Table 3 Statistics of the libraries used by the selected projects with number of libraries (# Lib.), number of classes (# Classes), number of methods (# Methods), number of public time related method (# PTM), hours of computation (Time), number of method calls to a Java time API (JC), and number of method calls to a library time method (LC).

Project	# Lib.	# Classes	# Methods	# PTM	Time (h)	JC	LC
ActiveMQ	546	76,142	674,692	154	4.685	1,099	0
Activiti	269	44,231	373,118	158	0.351	156	0
Airavata	168	19,871	182,982	117	0.125	274	0
Alluxio	298	45,254	426,670	179	0.307	338	0
Atmosphere	128	8,525	68,434	53	0.064	56	0
AWS-SDK-Java	161	18,803	173,239	126	0.172	234	0
Beam	782	299,928	2,903,163	29	20.161	123	0
Camel	2,611	612,571	6,540,417	4,474	45.420	585	0
Elastic-Job	145	20,888	205,733	100	0.196	53	0
Flume	522	113,117	1,086,062	414	1.258	293	0
Hadoop	431	66,282	571,628	319	0.624	2,798	0
Hazelcast	223	36,149	291,106	100	0.287	1,055	0
Hbase	429	80,956	765,446	188	5.316	1,642	0
Jetty	562	60,443	547,066	42	3.799	527	0
Kafka	37	7,401	68,511	34	0.139	417	0
Lens	452	119,486	1,270,709	593	0.972	152	0
NanoHTTPD	118	7,188	54,121	41	0.054	7	0
Neo4j	292	52,602	470,541	236	0.374	850	0
Sling	1,091	165,453	1,469,600	1,080	1.789	902	0
Twitter4j	92	15,293	186,570	105	0.126	66	0
Total	9,357	1,870,583	18,329,808	8,542	86.218	11,627	0

our hypothesis and we can conclude that developers mainly use local variables to implement time-related functionality.

5.4 RQ3 - What is the number of method calls to time methods in external libraries?

Our approach relies on a time semantic that we defined for the Java 8 time APIs. However, developers might also use time APIs provided by other libraries, such as Joda-Time.³ Furthermore, they might use library methods which wrap the Java 8 time APIs. This might impact the precision, recall, and finally also the applicability of our approach since it does not support these libraries and the extracted models would miss this time information. With JSR-310⁴, Java 8 improved the date and time APIs and we conjecture that developers rely only on them for handling time in their applications. With this research question, we want to verify our conjecture.

For each of the 20 projects in our data set, we performed the following steps: First, we collected the libraries used by the project using its build configuration. We erased the local repository used by Maven⁵ or Gradle⁶ and then executed the build process to download the libraries (*i.e.*, jar files) used to build the project. They are stored by the build system into the local Maven

³ <http://www.joda.org/joda-time/index.html>

⁴ <https://jcp.org/en/jsr/detail?id=310>

⁵ <https://maven.apache.org/>

⁶ <https://gradle.org/>

or Gradle repository. Since the build system also copies the jar files created during the build into the local repository, we manually removed them to keep only the jar files of the libraries.

Next, we used the Fernflower⁷ Java decompiler to reconstruct the source code of each retrieved library of the project. Then, we applied our approach to the decompiled source code to detect *public* methods of the categories RT and ET. To collect the list of a library’s time methods that encapsulate and export time functionalities, we have used the rules R_{rt} and R_{et} of our approach, presented in Section 3.1. With the two lists of Java 8 time methods and libraries’ time methods, we next ran our prototype tool on the source code of the project to detect calls to the Java 8 time methods and calls to the libraries’ time methods. To find out which concrete method of which class is called, we resolved the method bindings using the Eclipse JDT⁸ tools. The methods then were matched by their fully qualified name.

Table 3 presents descriptive statistics of the decompiled libraries. The number of libraries used by the 20 projects varies from 42 for Kafka to 2,611 for Camel, resulting in a total of 9,357 libraries to analyze. Decompiling the jar files of these libraries resulted in total in 1,308,284 Java source files implementing 1,870,583 classes and 18,329,808 methods. As expected, the source code decompiled from the libraries used by Camel contained the largest number of classes and methods, namely 612,571 classes and 6,540,417 methods. Applying our prototype tool to analyze the 1,870,583 Java classes and 18,329,808 methods required more than 86 hours of computation. More than half of it (45.42 hours) was spent on analyzing the methods contained by libraries used by Camel. The analysis produced in total a list of 8,542 public time methods which is 0.047% of the methods. Camel alone contains half of the public time methods.

The last two columns of the Table 3 report the results of analyzing the calls to the Java 8 timed methods (JC) and libraries time methods (LC). The values in the column LC are all 0, meaning that none of the 20 projects calls a method contained by a library that has been marked as time method by our semantics. This result clearly supports our conjecture and answers our research question RQ3: developers do not depend on methods in libraries to implement time-related functionality. They rely only on the Java time APIs.

5.5 RQ4 - Are the extracted models adequate to detect real errors in open source projects?

In addition to the quantitative evaluation, we also performed an initial assessment of the effectiveness of our approach to detect time related bugs. For this, we manually investigated the Jira issue tracker of the Apache Software Foundation seeking for bugs that involve time.

⁷ <https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine>

⁸ <https://www.eclipse.org/jdt/>

Table 4 List of time-related bugs taken from the Apache issue tracker. We present id, version affected, and a small summary of the issue description.

Issue ID	Version	Description
FLUME-1401	1.3.0	Asynchbase sink should be configurable to support timeout.
FLUME-3044	1.7.0	There are several method call in kafka sink with no timeout params, in some cases, kafka sink will await forever if no interruption.
HBASE-17341	2.0.0	In <i>ReplicationSource.terminate()</i> , a Future is obtained from <i>ReplicationEndpoint.stop()</i> . <i>Future.get()</i> is then called, but can potentially hang there if something went wrong in the endpoint <i>stop()</i> .
KAFKA-3540	0.10.1.0	Close the consumer, waiting indefinitely for any needed cleanup. That is not acceptable as it creates an artificial deadlock which directly affects systems that rely on Kafka A/I essentially rendering them unavailable.
KAFKA-4194	0.10.1.0	The loop below doesn't break when the timeout is reached, right? It will still block indefinitely until the metadata has been refreshed.
KAKFA-4306	0.10.1.0	If brokers are not available and we try to shut down connect workers, sink connectors will be stuck in a loop retrying to commit offsets forever.
LENS-1032	2.5.0-beta	We should provide option to kill the query upon timeout for users who are not interested in result beyond timeout.
LENS-1157	2.6	Execute with timeout fails to read metadata when user requests streamed results.

We searched for the keyword `timeout` and applied filters to return bugs only for Java projects reported between January 1st, 2016 and January 1st, 2018. We manually filtered the results removing the bug reports that were not dealing with timing issues in the source code. The filtering was necessary

because the majority of the bug reports returned by the query concern the adjustment of the timeout parameters for the integration test suite. Through this filtering, we obtained 8 reports, two from Flume, one from HBase, three from Kafka, and two from Lens. Note, 7 of the 8 bug reports come with an accepted patch that is attached to the report. At the moment of writing, one issue, namely FLUME-3044, was still open with a proposed patch attached to it. Table 4 presents the list of bugs and their short descriptions taken from their issue tracker summaries or github review comments.

Based on the bug descriptions, we divided the issues into two different types of errors:

- **No Time-Bound**: The method contains a loop that misses a time condition that sets a limit for its maximal execution time. The issues FLUME-3044, KAFKA-3540, KAFKA-4194, LENS-1032, and LENS-1157 belong to this category.
- **Indefinite Wait**: The method performs a call to an **EW** method, presented in Section 2.2, that potentially blocks the execution of a thread forever. The issues FLUME-1401, HBASE-17341, and KAFKA-4306 belong to this category.

We applied our approach to both, the original and patched versions of the code, and we ran our prototype tool to extract the timed automaton for each method that has been modified to fix the bug. Then, we used UPPAAL to first formally verify the existence of the reported bug and then its elimination in the patched version.

For the category **No Time-Bound**, we followed the methodology presented in Section 5.2 and created a set of ground truth transitions. For each method affected by this type of error, each researcher and the developer identified independently the transition that models the exit of the loop statement that causes the error. Then, we verified that the ground truth transition in the extracted automaton contains a time constraint that enforces a maximal execution time. Concerning the category **Indefinite Wait**, we used UPPAAL to verify that each automaton can always terminate by executing the formula $A \langle \rangle s_i$ where s_i is the state that identifies an ending state in the CFG of the method. The formula checks whether the state s_i can be eventually reached from the starting state. Our findings confirm the presence of all issues and furthermore the correctness of the proposed patches.

5.6 RQ5 - How much time and memory is required by our approach to extract the timed automata?

In this research question, we want to study how much time does our approach require to process the full source code of a project and how much time and memory it requires to extract the automaton for a single time method.

From a theoretical standpoint, the algorithm presented in Section 3 performs a greedy iteration over the statements of a method. The only exception

Algorithm 1: Pseudocode of our algorithm to refine a Timed Automaton

```

1 begin refine( $\mathcal{A}, T$ )
2   forall  $s \in \text{state}(\mathcal{A})$  do
3      $T_s \leftarrow \{t \in T \mid \text{contains}(t, s)\}$ 
4      $C \leftarrow \emptyset$ 
5     forall  $t \in T_s$  do
6       if  $\text{isET}(t)$  then
7          $\mathcal{A} \leftarrow \text{addETConstraint}(\mathcal{A}, s)$ 
8       else if  $\text{isEW}(t)$  then
9          $\mathcal{A} \leftarrow \text{addEWConstraint}(\mathcal{A}, s)$ 
10      else
11         $C \leftarrow C \cup \{t\}$ 
12     $\mathcal{A} \leftarrow \text{addTEConstraint}(\mathcal{A}, C, s)$ 
13 return  $\mathcal{A}$ 

```

is for statements that represent branching instructions, for which the algorithm iterates over the body of the branching instruction until a fix point is reached. Thus, the worst case scenario for this algorithm is a method in which every statement is a nested branching instruction. We denote n as the number of statements of the method and k as the length of the time information extracted, *i.e.*, the number of elements stored in the environment E . We can conclude that, in the worst case, the time complexity to extract the time information for a method is $\mathcal{O}(n!)$. The algorithm constructs the output, *i.e.*, list of the time information extracted, directly and no local memory is needed except for the fix point computation. Here, the algorithm needs to keep a copy of the environment computed in the previous iteration. Therefore, the space complexity to extract the time information from a method is $\mathcal{O}(k)$.

The algorithm to construct a timed automaton presented in Section 4 is divided into three different phases:

1. Construct an initial timed automaton.
2. Refine the automaton with time constraints.
3. Finalize the automaton with the runtime data.

In the first phase, the creation of the initial automaton is a one to one copy of the CFG representation into the UPPAAL automaton representation. Therefore, the creation of the automaton can be performed linearly in accordance with the size of the CFG. CFGs are a sparse type of graph since only maximum two transitions can depart from a node. We can conclude that the size of these graphs is bounded by $\mathcal{O}(n)$. This yields a time complexity of $\mathcal{O}(n)$ and a space complexity of $\mathcal{O}(1)$ for building the initial timed automaton.

We present the pseudo code of the second phase in Algorithm 1. The $\text{refine}(\mathcal{A}, T)$ function takes as input the automaton \mathcal{A} created in the previous phase and the extracted time information T . The function attaches to every state of the automaton \mathcal{A} its time constraint, if it exists. The retrieval

of the time information for each state is performed at line 3. This is achieved with a linear search on the list, yielding a time complexity of $\mathcal{O}(k)$. Then, for each time information of the state, the algorithm checks the type of time information and builds the correct time constraint for it, as presented in Section 4.2. Every statement between lines 6 and 11 is executed in $\mathcal{O}(1)$. In particular, if the time information t is needed to generate the Time Expired constraint, we save it in the set C . Once all the time information for the Time Expired constraint are collected, they are processed by the function in line 12. The function *addTEConstraint* adds the Time Expired constraints for the state s to the automaton \mathcal{A} . This is performed linearly in the size of C . Since each time information belongs to only one state, we can conclude that the worst case scenario is when all the time constraints are of type Time Expired. Thus, the time complexity for the function *refine*(\mathcal{A}, T) is the following:

$$\begin{aligned} T(n, k) &= \mathcal{O}(n) \times \left(\underbrace{\mathcal{O}(k)}_{\text{forall}} + \underbrace{\mathcal{O}(k)}_{\text{addTEConstraint}} \right) \\ &\quad \Big| \\ &= \mathcal{O}(nk) \end{aligned}$$

For what it concerns the space complexity, the worst case scenario is when C contains all the time information for every state, *i.e.*, $|C| = |T_s| = \mathcal{O}(k)$.

The last phase consists of replacing missing information with the data collected through the runtime monitoring. This is performed with a greedy pass over the automaton transitions. Depending on the type of the expression in the transition, a different strategy is used, as presented in Section 4.3. Both *Time Variables* and *RT Method calls* are performed with a space and time complexity of $\mathcal{O}(1)$. The last category of expression, *non-local time variables and unresolved method calls*, requires to process the list of time information recorded with the runtime monitoring. This step requires a scan of the list to identify those values that belong to the current considered transition. The worst case scenario is when every single time information extracted needs to be monitored at runtime, yielding a time complexity of $\mathcal{O}(nk)$ and a space complexity of $\mathcal{O}(k)$.

In summary, the time and space complexity to extract a timed automaton from a method with n statements and k time information are

$$\begin{aligned} T(n, k) &= \mathcal{O}(n^n) + \mathcal{O}(n) + \mathcal{O}(nk) + \mathcal{O}(nk) \\ &\quad \Big| \\ &= \mathcal{O}(n^n) + \mathcal{O}(nk) \\ \\ S(k) &= \mathcal{O}(k) + \mathcal{O}(1) + \mathcal{O}(k) + \mathcal{O}(k) \\ &\quad \Big| \\ S(k) &= \mathcal{O}(k) \end{aligned}$$

From the theoretical study of the space and time complexity, the time and memory required by our approach to produce a timed automaton heavily depends on the size and complexity of the source code implementing a method. Therefore, to empirically study the space and time complexity we consider the following three code metrics: (i) number of statements, (ii) McCabe cyclomatic

Table 5 Minimum, median, and maximum values of the source code metrics computed for the 11, 772 time methods of the 20 open source Java projects with number of statements (# Statements), McCabe Cyclomatic Complexity (CC), and number of variables (# Variables).

Project	# Statements			CC			# Variables		
	min	median	max	min	median	max	min	median	max
ActiveMQ	1	19	182	1	3	61	1	18	136
Activiti	1	17	282	1	4.5	86	2	13.5	72
Airavata	1	19.5	390	1	6	46	4	19	73
Alluxio	1	8	111	1	2	46	1	19	62
Atmosphere	1	17	60	1	3	19	8	18.5	67
AWS-SDK-Java	1	9	90	1	2	15	2	11	57
Beam	1	13	61	1	4	21	2	22.5	57
Camel	1	11	113	1	2	43	1	11	316
Elastic-Job	3	6	26	1	3	6	5	10	22
Flume	1	19	108	1	3	58	5	20	109
Hadoop	1	16	437	1	3	81	2	23	131
Hazelcast	1	10	58	1	2	18	1	16	294
Hbase	1	18	192	1	4	81	2	26	220
Jetty	1	19	520	1	2	113	4	18	103
Kafka	1	12	110	1	3	42	1	26.5	312
Lens	1	16	105	1	3	27	5	18	96
NanoHTTPD	1	8	57	1	1	18	5	8	41
Neo4j	1	9	77	1	2	24	2	14	102
Sling	1	14	162	1	3	47	2	17.5	74
Twitter4j	1	9.5	109	1	2	42	5	16	65
Overall	1	14	520	1	3	113	1	19	316

complexity (CC), and (iii) number of variables used in the method. The number of statements indicates the size of a method. The cyclomatic complexity is a measure of the number of linearly independent paths through a program. We included it because the implementation of our approach iterates over branching instructions. Finally, we selected the number of variables as metric because our implementation keeps track of all variables in the environment and adds a special flag to the time related ones. Furthermore, when our implementation analyzes a branching statement, it creates a copy of the environment which is a time and memory consuming operation. We have computed these metrics based on the abstract syntax tree representation of the source code and they are available in our prototype tool.

Table 5 shows the minimum, median, and maximum values for the three code metrics measured for the 11, 772 time methods per project. Statistics on the overall size of the projects are presented in Table 2. Over all projects, the time methods have a median size of 14 statements and a maximum size of 520 statements. The median cyclomatic complexity is 3 and the maximum complexity is 113. The longest and most complex time method is found in the Jetty project. The time method using the highest number of variables is found in the Camel project. Overall, the number of variables ranges from 1 to 316 with a median value of 19 variables used per method.

We have performed our experiment to measure the time and memory consumption of our approach on a MacBook Pro with a 2.5GHz Intel i7 CPU, 16GB of main memory, running macOS 10.13.6. The two columns on the left hand-side of Table 6 with the heading Full Project show the total time to process all methods of each project. Since our approach consists of two phases, we

Table 6 Time required to parse (Parsing) and extracting the timed automata (TA), along with the minimum, median, and maximum amount of Time and Memory required by our prototype tool to generate a timed automaton.

Project	Full Project (s)		Time (ms)			Memory (KB)		
	Parsing	TA	min	median	max	min	median	max
ActiveMQ	158	381	133	229	816	6,715	9,066	54,540
Activiti	434	962	106	139	319	6,518	7,085	54,170
Airavata	645	687	112	211	447	5,422	12,090	31,720
Alluxio	154	291	122	171	401	6,035	8,523	40,070
Atmosphere	23	81	136	182	392	7,225	9,056	22,910
AWS-SDK-Java	20,635	6,846	107	151	352	5,353	7,125	28,540
Beam	165	207	116	166	345	6,967	7,760	18,410
Camel	4,071	1,918	105	145	471	3,791	10,160	31,350
Elastic-Job	16	37	107	115	147	7,223	7,657	10,160
Flume	36	79	125	211	376	7,306	9,300	18,920
Hadoop	1,379	1,657	113	233	2,857	3,041	10,220	89,060
Hazelcast	524	1,068	102	185	442	5,231	9,740	26,030
Hbase	1,394	924	114	212	808	5,122	9,359	51,850
Jetty	228	423	111	185	466	6,869	10,110	32,160
Kafka	90	328	107	176	336	6,339	10,020	34,510
Lens	56	193	106	165	503	6,707	9,876	31,280
NanoHTTPD	3	10	112	123	247	7,567	7,763	25,110
Neo4j	589	981	98	144	453	5,393	7,264	34,010
Sling	751	762	109	157	1,410	5,975	8,749	28,370
Twitter4j	24	91	121	170	302	7,389	9,233	15,190
Overall	31,374	17,926	98	192	2,857	3,041	9,549	89,060

first computed the time required (using Eclipse JDT) to parse the source code into an Abstract Syntax Tree. Depending on the project size, our approach required from 3 seconds for parsing the source code of NanoHTTPD to more than 5.73 hours for parsing the source code of AWS-SKD-Java. Not surprisingly, the projects with the highest number of methods (see Table 2) also took the longest time to parse. For most of the projects, namely 16, the parsing took less than 13 minutes. Regarding the second phase, the extraction of the timed automata, the results are presented in the column TA. For 16 projects, our approach extracted the timed automata in less than 17 minutes. Also for this phase NanoHTTPD took the least amount of time, namely 10 seconds, while AWS-SKD-Java took 1.9 hours.

Furthermore, since our approach computes the time constraints for only time methods, we also computed the time and memory used to produce the timed automata for these methods. We report the minimum, median, and maximum amount of time and memory per project in the columns on the right hand side of Table 6. In addition, we also show the distributions of the time and memory consumption with violin plots in Figure 5. Note, these results comprise both, the parsing and the extraction of the timed automaton for a given time method. Looking at the table and the violin plots, we can see that the time required to process a method varies from 98ms to 2857ms with a median value of 192ms using between 3 and 89MB of memory with a median value of 9.5MB. With respect to the median time, the time methods in the Hadoop project took the longest to process, namely 233ms, while the time methods in the Elastic-Job project took the least amount of time, namely 115ms. The maximum amount of time to process a time method was in the Hadoop project,

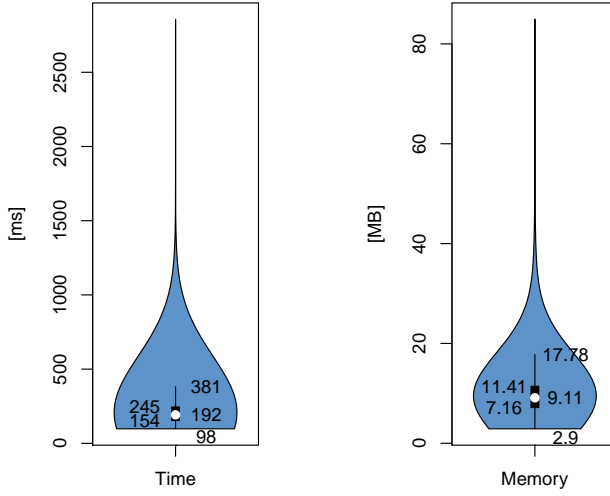


Fig. 5 Violin plots with the distribution and the quartiles of time and memory consumption.

namely 2857ms. The violin plot for Time in Figure 5 shows that for 93.25% of the time methods, our tool required less than 381ms. With respect to the median memory consumption, the processing of the time methods in Airavata consumed the highest amount of memory, namely 12MB. Instead, the time methods in Hadoop required the minimum but also the maximum amount of memory with 3MB and 89MB, respectively. Looking at the memory violin plot in Figure 5, 95.37% of time methods required less than 18MB of memory to be processed by our prototype tool.

In addition to the distribution of time and memory, we also investigated the correlation between the three code metrics and time and memory consumption. Since none of our five metrics have a normal distribution, we used the *Spearman's rank* method to compute the correlation. A Spearman ρ value of +1 and -1 indicates high positive or high negative correlation, whereas 0 indicates that the variables under analysis do not correlate at all. Values greater than +0.3 and lower than -0.3 indicate a moderate correlation; values greater than +0.5 and lower than -0.5 are considered to be large/strong correlations [33]. The strongest correlation is shown between the cyclomatic complexity and the time, with a ρ of 0.54. This means an increase in the complexity of a method most likely leads to an increase in the time needed to extract the timed automaton for that method. Instead, the cyclomatic complexity shows only a weak correlation with the memory consumption, namely 0.26. The number of variables moderately correlates with both, time and memory, with a Spearman's ρ of 0.48. Finally, also the number of statements moderately correlates with the time and the memory consumption with a ρ of 0.38 and 0.36, respec-

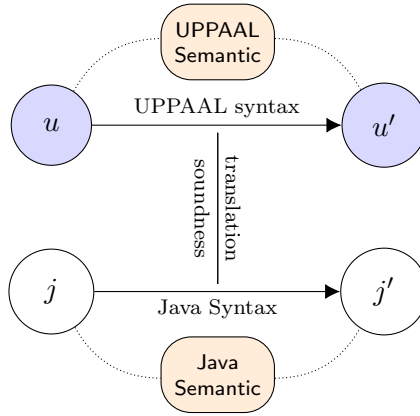


Fig. 6 Sketch of the proof of soundness of our translation from Java source code to UPPAAL.

tively. All the correlation coefficients are significant with a p-value lower than 0.01.

We can conclude that our approach can extract the timed automata for the majority of projects in less than half an hour. Furthermore, if a developer narrows down the scope to a specific method, our approach performs the extraction in less than 381ms using roughly 18MB of memory in 95% of the cases. The cyclomatic complexity of the method impacts the processing time the most, followed by the number of variables. The latter is also the factor that most likely impacts the memory consumption.

6 Discussion

This section discusses the results and their implications on research and practice. Furthermore, we discuss the limitations of our current approach and the potential threats to validity of our empirical findings.

6.1 Summary of Results

In Section 3, we proposed a time semantics for Java that we used to develop an approach to automatically extract timed automata from the source code of a Java method. We presented this approach in Section 4 and its evaluation in Section 5.

A key property of an automated approach is that it should be sound and complete. We investigated this property with our research questions RQ1, RQ2, and RQ3. The results of research question RQ1 show a precision and recall for our time semantics of 100%. The results of research question RQ2 show that, for 95% of the time methods, our approach could extract timed automata that model the *entire* time behavior of those methods. Only for 5%

of the methods a dynamic analysis step was needed to add the missing time information that could not be extracted or modeled through our static analysis approach. But, also this step is automated by our approach and does not need any human intervention. The shortcoming, however, is that the created instances of the timed automata for these methods do not cover all possible values for the time variables used in a method but only the values created through running the test cases. Consequently, while only affecting 5% of the methods, our translation from Java source code to UPPAAL is not complete.

Concerning the soundness, RQ1 shows a perfect precision and recall that supports a sound translation. The diagram in Figure 6 presents a sketch of the proof of soundness of our translation. In fact, the formal proof assures that any time-transition (j to j') of a given Java program has the equivalent time semantics applied by UPPAAL in processing the transition from the respective state u to state u' . In other words, we assure that our UPPAAL correctly simulates any valid transition of its Java counterpart w.r.t. time. In principle, the proof of soundness is a structural induction proof based on the Java statement that involve time, whose further details are beyond the scope of this paper.

The results of research question RQ3 clearly show that developers exclusively use the Java 8 time APIs to implement time-related functionality. They do not use methods provided in third party libraries that are not covered by our time semantics. Summing up the results of RQ1, RQ2, and RQ3, we view our approach to be sound and complete, at least with respect to our time semantics for the version 8 (and later versions) of the Java time APIs. The results of research question RQ4 show that our extracted timed automata can be effectively used to detect time related errors and to confirm the correctness of the proposed patches. Finally, from the results of research question RQ5 we can conclude that the runtime complexity of our approach scales with the size and complexity of the project, requiring 30 minutes to process most of the projects.

6.2 Implications of Results

Concerning the implications on the research in this area, our approach improves existing approaches that model time as a sequence of events represented in a tree-like structure, such as presented by Walkinshaw et al. in [59]. These approaches fail to model time aspects of a program, such as timing delays. Furthermore, our approach does not require developers to *manually* craft a timed automaton to formally verify the time behavior of a method, such as presented by Jayaraman et al. in [35]. This prevents the introduction of human errors.

The formal definition of a semantics for the time domain can be used by other formal verification tools, such as Java Path Finder [30], to verify a richer spectrum of properties, such as concurrency defects like deadlocks, and unhandled exceptions like `NullPointerExceptions` and `AssertionErrors`. In

addition, our semantics can be used to investigate techniques that can help developers to find time related problems in early phases of the development. We recently presented an approach [44] that uses our time semantics encoded in the Z3 SMT solver [19, 7] to identify values for time variables that cause a program to fail.

Our results also have several implications on practitioners. Software developers and testers can use our approach to *automatically* extract timed automata to verify the implementation of time-related functionality in Java programs. Developers can specify time properties that must hold during the development of the system and through that discover errors in the implementation of time-related functionality of modified Java methods. Currently, our implementation does not exploit multi-processors capabilities and the total time to process a project can be further reduced parallelizing the analysis. However, on our laptop machine the process of a project required half an hour with low memory footprint. Therefore, developers could integrate our approach into the pipeline of their continuous integration systems to analyze each new version of a program committed to the source code repository. Furthermore, the extracted timed automata can be used during code reviews to help the reviewers analyze the correctness of the implementation. For instance, reviewers can run queries with the UPPAAL model checker to verify that the implementation satisfies the specification.

6.3 Limitations

In our manual analysis of the Java 8 APIs presented in Section 2.2, we discovered four different categories of time methods. Currently, we do not provide any abstraction that can model methods of the category **ST** where a method call has a time behavior if and only if a timeout is set by a preceding method call. For instance, the `connection` method of the `URLConnection` class by default has no upper time limit for establishing the connection. Instead, a timeout can be set by calling the `setConnectTimeout` method before calling the `connection` method. We plan to address this limitation in our future work.

While we consider **EW** methods of the Java 8 time APIs, we do not gather the project's **EW** methods. Although they are time related methods, they do not reference any time variable and currently, we do not provide any rule that infers them from the statical analysis of the source code. Future work will be devoted to infer this kind of time related methods through abstract interpretation or dynamic analysis to discover patterns that identify methods which can possibly block the execution of a program forever. Furthermore, we model **EW** method calls with a self-loop without considering its synchronization with external events. This limitation could be overcome by considering a network of timed automata. Due to the inheritance and polymorphism offered by Java, multiple automata can send a message to the **EW** state. Considering all possibilities is not a valid option since it may not be representative of the implementation. We plan to address this limitation complementing our current

approach with a dynamic analysis that monitors the program and identifies the sender and receiver instructions. Then, we could simulate this behavior in the network of timed automata forcing the corresponding states to communicate via channels.

Currently, our approach requires a developer to manually specify which property must hold in the extracted automata. Once she/he has defined them, our approach can automatically verify those. We plan to address this limitation in our future work by automatically inferring the properties to verify from the test suite of a project.

6.4 Threats to Validity

In the following, 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 internal validity concerns the reliability of our prototype implementation. We mitigated this threat by testing the prototype tool manually and with unit tests. For each analysis in RQ1 and RQ2, we randomly selected 400 methods that were used to evaluate our approach. The size of our sample set is larger than the minimum number (372) required to obtain results at a 95% confidence level with a 5% margin of error. Moreover, we showed the applicability of our approach on 8 bugs taken from Apache open source projects (RQ4). Our approach confirmed the existence of the bugs and also validated the correctness of the proposed patches.

Another threat concerns the possibility that our approach might miss to model the behavior stemming from time methods that are provided or wrapped by third party libraries. With the research question RQ3 we evaluated the impact of this threat. We set up an empirical study with 20 open source Java projects where we investigated the number of public methods of the projects' libraries that implement time-related functionality and are called by methods in the 20 Java projects. The results of our study show that only 0.047% of the methods in the libraries export time functionalities. Moreover, none of them is used in the source code of the 20 projects. We therefore can conclude that developers do not depend on methods in libraries to implement time-related functionality but they rely only on the Java time APIs that our approach extracts with 100% precision and recall.

The manual analysis to create the ground truth for answering RQ1 and RQ4 poses another threat to internal validity. We mitigated it by asking also an independent developer to perform the manual analysis. Each discrepancy then has been discussed with all participants until a consensus was reached.

Java does not permit to directly allocate or free memory. This could introduce a threat to the validity of the results of RQ5. We mitigated this threat by calling the Garbage Collector (GC) before applying our approach to remove the extra memory allocated by the JVM. Since the call to the GC is async, we poll the method call until it is eventually invoked by the JVM. We have

used the APIs offered by the `ManagementFactory` class to monitor when the release of the spurious memory is indeed executed.

External Validity. Threats to external validity concern the generalization of the results to other software projects. We mitigated this threat by choosing 20 open source Java projects that differ in size and domains to improve the generalization of our results. We also implemented our approach in a prototype tool that is publicly available online¹ and can be applied to other Java projects to extend our studies. Furthermore, our formal time semantics can be adapted to other programming languages, such as C#, that use a semantics of time similar to Java.

Furthermore, in RQ2 we relied on Daikon to extract invariants that identify pure methods w.r.t. time. We configured Daikon to monitor the execution of the tests provided with the projects. Its results heavily depend on the characteristics of the test suite and, therefore, they might not be representative for every possible input value of the tested methods. However, tests cover how a specific piece of code should be executed and the results extracted by Daikon can be considered representative use cases for the method under analysis. Further studies are required to additionally verify this generalization, complementing Daikon with static approaches and with automatic unit test generators, *e.g.*, Randoop [49], to execute the method with additional input.

7 Related Work

One of the main contributions of this paper is our time semantics that we have used to determine which Java statements are time related. In the domain of semantics for the Java programming language, Bogdanas and Roşu [14] present a formal semantics for Java version 1.4 based on their \mathbb{K} -Framework [53]. They formalize the language syntax and how the Java Virtual Machine (JVM) interprets the bytecode. Similarly, the Real-Time for Java Expert Group [15] provide a specification, called RTSJ, that enforces a specific semantics for the Java Virtual Machine and introduces a new set of APIs. They specify how the JVM should interpret specific classes to enable the creation, verification, analysis, execution, and management of Java threads for real time programs. However, their focus is on the scheduling of threads for reaching a real-time execution and not on verifying the time-related functionality implemented in a program.

The second contribution of this paper is an approach to extract timed automata from source code. There are existing works that translate source code to timed automata. Cicirelli et al. [18] present a library for UPPAAL that is able to reproduce the semantics of major Java concurrent and synchronization mechanisms. Yang et al. [61] present a tool that translates Simulink Stateflow models into UPPAAL timed automata for verification. With the verification power of UPPAAL, their approach manages to find design defects that were missed by the Simulink Design Verifier. Timed automata are also used as basic design specification to verify properties in a program. The approach presented

by Jayaraman et al. [35] takes as input a network of timed automata provided by developers. The network is used as base knowledge of the specific real-time system. They monitor the execution of a program verifying that its behavior conforms to the network of timed automata provided. In an analogous way, Hakimipour et al. [27] propose a technique to automatically generate a program from timed automata. They use a timed automaton to produce an RTSJ program that is executable on single- and multi-processor platforms. Georgiou et al. [25] present a technique to translate timed input/output automata into distributed executable Java programs. These approaches require developers to manually construct such time models which could introduce errors. Furthermore, these approaches do not guarantee that the models are an exact representation of the implementation. Our approach, on the contrary, automatically extracts models from the implementation that are directly amenable to the verification.

In the domain of verifying properties of source code, NASA developed Java Path Finder [30], a framework for verification and debugging of Java programs. The tool is used to verify properties of Java programs with a focus on race conditions. It converts the bytecode of a program into the Promela model checking language. Similarly, Henzinger et al. [32], present a framework for verifying properties of C programs for the mutex API. Bandera [28] automatically extracts a state machine from Java source code amenable to verification. In all the previous approaches, the models represent the control and data flow of a program without taking into account the specifics of the domain of time. The work of Walkinshaw [59] describes an extension of an existing state machine inference technique in which it accounts for temporal properties of the subjected system. However, it does not consider time explicitly but only represents it as a sequence of events as they happened in the execution of a program. On the contrary, our approach considers the specific time semantics of the programming language that models more time properties, such as timing delays.

8 Conclusions

In this paper, we presented an approach to automatically extract timed automata from the source code of a Java program. We first introduced a definition of the semantics for time in the Java programming language. The time semantics is then used by our approach to extract the timed automata. Our approach first identifies Java statements related to time, second, it infers time constraints, and lastly, it uses this information to create the timed automata.

We have implemented our approach in a prototype tool that was used to answer five research questions. First, we empirically studied the soundness and completeness of our time semantics manually analyzing 400 Java methods randomly selected from 20 open source Java projects. The results of the manual evaluation show that our time semantics can detect time statements and time variables with 100% precision and recall (see Subsection 5.2). In the sec-

ond research question we studied the ratio of methods for which our approach could extract a timed automaton that entirely models the time behavior. The results show that this worked for 95% of the methods. For the remaining 5% of methods our approach performed an additional dynamic analysis step to create an instance of the timed automaton for each test case that then can be used for verification (see Subsection 5.3). With the third research question we found that our time semantics is adequate to describe all the time APIs that are used by developers and it does not need to consider time methods defined in external libraries (see Subsection 5.4). Furthermore, with the fourth research question, we presented 8 examples of real bug reports of four Apache projects. We used our approach to verify the existence of the bugs as well as the correctness of their patches (see Subsection 5.5). With the last research questions, we showed that the median time for extracting a timed automaton is 0.192 seconds and the total time to process most of the projects is 30 minutes (see Subsection 5.6). Our results benefit researchers and practitioners. In addition to automating the extraction of timed automata, our time semantics can be used by other formal verification tools to verify a richer spectrum of properties. Furthermore, developers can benefit from our research, for instance by integrating our approach into their continuous integration system to verify that the time specification is correctly implemented.

Future work will be dedicated to improve our approach to provide a model also for methods of the category **ST**. [Moreover, we plan to extend our prototype tool to support annotations that allow developers to manually mark the time category of their methods.](#) Currently, developers have to encode their requirements in UPPAAL. Future work will be devoted to remove this manual step and infer them automatically from the test suite of their projects. Furthermore, we plan to extend our approach to other programming languages, such as C#. And finally, we will extend our studies to cover a larger set of open source projects and also industrial projects. Concerning industrial projects, we also plan to evaluate the usefulness of our approach with software developers and testers, and apply our approach to real-time systems.

References

1. Abadi, M., Cardelli, L.: A theory of objects. Springer Science & Business Media (2012)
2. Abrahamson, J., Beschastnikh, I., Brun, Y., Ernst, M.D.: Shedding light on distributed system executions. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 598–599. ACM (2014)
3. Alur, R.: Timed automata. In: International Conference on Computer Aided Verification (CAV), pp. 8–22. Springer (1999)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science **126**(2), 183–235 (1994)
5. Baier, C., Katoen, J.P., Larsen, K.G.: Principles of model checking. MIT press (2008)
6. Baliga, A., Ganapathy, V., Iftode, L.: Detecting kernel-level rootkits using data structure invariants. IEEE Transactions on Dependable and Secure Computing **8**(5), 670–684 (2011)
7. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England), vol. 13, p. 14 (2010)

8. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: *Formal Methods for the Design of Real-Time Systems*, pp. 200–236. Springer (2004)
9. Bengtsson, J., Griffioen, W.D., Kristoffersen, K.J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Verification of an audio protocol with bus collision using UPPAAL. 1102, pp. 244–256. Springer-Verlag (1996). URL <http://www.docs.uu.se/docs/rtmv/papers/bgkllpw:cav96.ps.gz>
10. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A.: Inferring models of concurrent systems from logs of their behavior with CSight. In: *Proceedings of the 36th International Conference on Software Engineering*, pp. 468–479. ACM (2014)
11. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A., Anderson, T.E.: Mining temporal invariants from partially ordered logs. *ACM SIGOPS Operating Systems Review* **45**(3), 39–46 (2012)
12. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 267–277. ACM (2011)
13. Beschastnikh, I., Wang, P., Brun, Y., Ernst, M.D.: Debugging distributed systems. *Queue* **14**(2), 50 (2016)
14. Bogdan, D., Roşu, G.: K-Java: A complete semantics of Java. In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 445–456. ACM (2015)
15. Bollella, G., Gosling, J.: The real-time specification for Java. *Computer* **33**(6), 47–54 (2000)
16. Börger, E., Fruja, N.G., Gervasi, V., Stärk, R.F.: A high-level modular definition of the semantics of C#. *Theoretical Computer Science* **336**(2-3), 235–284 (2005)
17. Cardelli, L., Gordon, A.D.: Mobile ambients. In: *International Conference on Foundations of Software Science and Computation Structure*, pp. 140–155. Springer (1998)
18. Cicirelli, F., Furfaro, A., Nigro, L., Pupo, F.: Modelling Java concurrency: an approach and a UPPAAL library. In: *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1373–1380. IEEE (2013)
19. Deharbe, D., Fontaine, P., Paleo, B.W.: Quantifier inference rules for SMT proofs. In: *First International Workshop on Proof eXchange for Theorem Proving (PxTP)* (2011)
20. Dijkstra, E.W.: The humble programmer. *Communications of the ACM* **15**(10), 859–866 (1972)
21. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* **69**(1-3), 35–45 (2007)
22. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: *International Conference on Computer Aided Verification*, pp. 501–505. Springer (2004)
23. Filaretto, D., Maffei, S.: An executable formal semantics of PHP. In: *European Conference on Object-Oriented Programming*, pp. 567–592. Springer (2014)
24. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 171–183. ACM (1998)
25. Georgiou, C., Musial, P.M., Ploutarchou, C.: Tempo-toolkit: Tempo to Java translation module. In: *International Symposium on Network Computing and Applications (NCA)*, pp. 235–242. IEEE (2013)
26. Guth, D.: A formal semantics of Python 3.3 (2013)
27. Hakimipour, N., Strooper, P., Wellings, A.: TART: Timed-automata to real-time Java tool. In: *International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 299–309. IEEE (2010)
28. Hatcliff, J., Dwyer, M.: Using the Bandera tool set to model-check properties of concurrent Java software. In: *International Conference on Concurrency Theory (CONCUR)*, pp. 39–58. Springer (2001)
29. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: *ACM SIGPLAN Notices*, vol. 50, pp. 336–345. ACM (2015)

30. Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)* **2**(4), 366–381 (2000)
31. Hennessy, M.: Algebraic theory of processes. MIT press (1988)
32. Henzinger, T.A., Necula, G.C., Jhala, R., Sutre, G., Majumdar, R., Weimer, W.: Temporal-safety proofs for systems code. In: *International Conference on Computer Aided Verification (CAV)*, pp. 526–538. Springer (2002)
33. Hopkins, W.G.: A new view of statistics. Internet Society for Sport Science (2014)
34. Hune, T., Larsen, K.G., Pettersson, P.: Guided synthesis of control programs using UPPAAL. *Nord. J. Comput.* **8**(1), 43–64 (2001)
35. Jayaraman, S., Hari, D., Jayaraman, B.: Consistency of Java run-time behavior with design-time specifications. In: *International Conference on Contemporary Computing (IC3)*, pp. 548–554. IEEE (2015)
36. Khan, M.T., Serpanos, D., Shrobe, H.: ARMET: Behavior-based secure and resilient industrial control systems. *Proceedings of the IEEE* **106**(1), 129–143 (2018)
37. Laporte, C.Y., Nabil, B., Mikel, D.: Measuring the cost of software quality of a large software project at bombardier transportation: a case study. *Software Qual. Manage* **14**(3), 14–31 (2012)
38. Larsen, K.G., Pettersson, P., Yi, W.: Diagnostic model-checking for real-time systems. In: *Proc. of Workshop on Verification and Control of Hybrid Systems III*, 1066, pp. 575–586. Springer-Verlag (1995)
39. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)* **1**(1-2), 134–152 (1997)
40. Le, T.D.B., Le, X.B.D., Lo, D., Beschastnikh, I.: Synergizing specification miners through model fissions and fusions (t). In: *Automated Software Engineering (ASE)*, 2015 30th IEEE/ACM International Conference on, pp. 115–125. IEEE (2015)
41. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (t). In: *Automated Software Engineering (ASE)*, 2015 30th IEEE/ACM International Conference on, pp. 81–92. IEEE (2015)
42. Lindahl, M., Pettersson, P., Yi, W.: Formal design and analysis of a gear-box controller. In: *Proc. of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, no. 1384 in *Lecture Notes in Computer Science*, pp. 281–297. Springer-Verlag (1998)
43. Liva, G., Khan, M.T., Pinzger, M.: Extracting timed automata from Java methods. In: *Proceedings of the IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 91–100 (2017)
44. Liva, G., Khan, M.T., Spegni, F., Spalazzi, L., Bollin, A., Pinzger, M.: 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)
45. Lo, D., Mariani, L., Pezzè, M.: Automatic steering of behavioral model inference. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pp. 345–354. ACM (2009)
46. McCarthy, J.: A formal description of a subset of ALGOL. Tech. rep., Stanford University, Department of Computer Science (1964)
47. McCarthy, J.: Towards a mathematical science of computation. In: *Program Verification*, pp. 35–56. Springer (1993)
48. Milner, R.: Communication and mobile systems: the π -calculus. Cambridge Uni (1999)
49. Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. In: *ECOOP 2005 — Object-Oriented Programming*, 19th European Conference, pp. 504–527. Glasgow, Scotland (2005)
50. Park, D., Ștefănescu, A., Roșu, G.: KJS: A complete formal semantics of Javascript. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’15)*, pp. 346–356. ACM (2015)
51. Plotkin, G.D.: LCF considered as a programming language. *Theoretical computer science* **5**(3), 223–255 (1977)
52. Plotkin, G.D.: A structural approach to operational semantics. Computer Science Department, Aarhus University Denmark (1981)

53. Roşu, G., Serbănută, T.F.: An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* **79**(6), 397–434 (2010)
54. Sangiorgi, D., Walker, D.: *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press (2003)
55. Schiller, T.W., Ernst, M.D.: Reducing the barriers to writing verified specifications. *ACM SIGPLAN Notices* **47**(10), 95–112 (2012)
56. Stärk, R.F., Schmid, J., Börger, E.: *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media (2012)
57. Steel, T.: A formalization of semantics for programming language description. In: *Proc. IFIP WG Formal Language Description Languages for Computer Programming*, pp. 25–36 (1969)
58. Tristan, J.B.: *Formal verification of translation validators*. Ph.D. thesis, Université Paris 7 Diderot (2009)
59. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 248–257. IEEE Computer Society (2008)
60. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. *Information and computation* **115**(1), 38–94 (1994)
61. Yang, Y., Jiang, Y., Gu, M., Sun, J.: Verifying Simulink Stateflow model: timed automata approach. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 852–857. ACM (2016)
62. Yovine, S.: Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer* **1**(1-2), 123–133 (1997)