2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation

Extracting Timed Automata from Java Methods

Giovanni Liva, Muhammad Taimoor Khan, and Martin Pinzger
Software Engineering Research Group,
Alpen-Adria Universität Klagenfurt, Austria
Email: {giovanni.liva, muhammad.khan, martin.pinzger}@aau.at

***Abstract*—The verification of the time behavior in distributed, multi-threaded programs is challenging, mainly because modern programming languages only provide means to represent time without a proper semantics. Current approaches to extract time models from source code represent time only as a sequence of events or require developers to manually provide a formal model of the time behavior. This makes it difficult for developers to verify various aspects of their systems, such as timeouts, delays and periodicity of the execution.**

In this paper, we introduce a definition of the time semantics of the Java programming language. Based on the semantics, we present an approach to automatically extract timed automata and their time constraints from the Java methods source code. First, we detect Java statements which involve time, from which we then extract the timed automata that are directly amenable to the verification of time properties of the methods.

We evaluated the accuracy of our approach on ten open source Java projects that heavily use time in their source code. The results show a precision of 98.62% and recall of 95.37% in extracting time constraints from Java code. Finally, we demonstrate the effectiveness of our approach with five reported bugs of four different Apache systems that we could confirm.

I. Introduction

Understanding programs that are large, distributed and multi-threaded consume a large amount of developer's time. As presented by Dano et al. [1], reverse engineering a multi-threaded program is, using their analogy, *a trip down to the Hades*. Other studies [1], [2] show that comprehension activities alone consume about $40\% - 60\%$ of all the available resources. This is mainly due to two reasons: (i) high staff-turnover, and (ii) the frequent evolution of programs makes it complicated to keep documentation up to date. Therefore, an automated technique to reverse engineering models from source code helps to understand and document multi-threaded or distributed programs. A key aspect of such systems is the usage of time, with integer values, to bound the waiting time of when certain events may occur. The time is also used to set execution timeouts or schedule events to occur periodically. A common technique to model the time behavior and the time constraints of these systems is the timed automata formalism.

The verification of time properties of the programs, modeled as timed automata, have been intensively studied over the last 20 years. Timed automata is a formalism introduced by Alur [3] that allows to model and analyze the timing behavior of multi-threaded programs. For instance, Jayaraman et al. [4] used the timed automata modelled by the developers to monitor the subject system and to verify whether its execution conforms to its specification. Hakimipour et al. [5] present an approach to derive real-time Java programs from the timed automata created by the users. Following the same idea, Georgiou et al. [6] present a technique to translate timed input/output automata into executable Java code.

Some recent approaches [7], [8] allow to extract a time model automatically from the Java source code. However, these approaches do not consider the semantics of the language and fail to model the real-time domains that are involved in the program, such as the time. They only retain time as a sequence of events represented in a tree-like structure and fail to model other time aspects of the program, such as timing delays. Also, such models assume that a program execution can be completely modeled as a sequence of states or system events. These approaches have mainly two common shortcomings: (i) they are based on informal semantics of time and thus are not reliable, failing to detect the semantic inconsistencies in the programs and (ii) the model is manually developed and thus, it is error prone.

In contrast to the afore-mentioned approaches, we present an approach that automatically extracts timed automata from the Java methods source code based on our definition of time semantics, which is directly amenable to verification. Algorithm 1 shows the steps of our approach to extract the timed automata. Based on our semantics of time, first we parse a Java method and extract its time related statements. Second, we map the extracted time statements into time constraints. Finally, we extract states and transitions creating the timed automata. The extracted timed automata permits to assure that the methods contain a correct representation of time according to the expected behavior. The timed automata extracted are not directly amenable to the verification because they contain expressions whose values are not known at compile time. We infer and replace these values using dynamic analysis to monitor the expressions during the execution of various test cases. For each run of a test case we obtain a set of instances of the timed automaton that can be used for verification of safety and liveness properties, *e.g.*, termination.

We manually evaluate the precision and recall of our approach in extracting timed automata from the source code using a random sample of 400 Java methods. The results show a precision of 98.62% and a recall of 95.37%.

Overall, the main contributions of this paper are:

- a semantics of the time for Java;

- an approach to automatically extract timed automata for Java methods;
- and an evaluation of our approach with a random sample of 400 Java methods.

The remaining of the paper is organized as follows: Section II gives the time semantics for a subset of Java 8 methods and Section III presents our approach for extracting timed automata from Java methods. Section IV presents the evaluation and the discussion of the results are in Section V. Section VI presents the related work. Finally, we conclude the paper in Section VII.

II. Semantic of Time in Java

Modern programming languages, such as Java or C#, provide the possibility to explicitly represent time with built in APIs, such as Java's `java.time.LocalDate` or, *implicitly*, using integer values from the domain of $\mathbb{Z}$. Therefore, these languages fail to detect time related semantic inconsistencies in the programs and, it results in programs failure or preemptive termination. For instance, a Java program with the statement `long now = startingTime - 10000`, is syntactically correct and compiles but it may throw an `IllegalArguementException` at run-time when the variable `now` is used in a time method call. This could happen because the Java compiler does not understand the semantics of time and thus fails to prevent the variable `now` to be negative.

Recently, Bogdanas and Roşu [9] present a complete semantics definition for the Java 1.4 language, without considering time. We address this limitation defining a time semantics based on the *implicit* representation of the time in Java programs. This is a pre-requisite of our analysis and it is presented in the following section.

A. Time Related Java 8 Methods

As a first step, we manually analyzed the API documentation of Java 8 SDK classes in the packages `java.time`, `java.net`, `java.util`, and `java.lang` identifying the methods that deal with time. From this analysis, we define four categories of methods which involve time:

1) **Return Time:** The first category covers methods that return an integer value that represents time, *e.g.*, the static `currentTimeMillis()` method of the `System` class returns the current time in milliseconds.
2) **Explicit Timeout:** The second category covers methods that contain a time parameter. For instance, the `connect` method of the `Socket` class, if called with two parameters, the second parameter specifies the maximum amount of time for making a connection.
3) **Explicit Wait:** The third category of 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:** The last category is composed of methods which change their behavior with a time constraint that is set by a preceding method call. For instance, the `connection` method of the `URLConnection` class by default has no upper limit for making a connection. Instead, a timeout can be set by calling the `setConnectTimeout` method before calling the `connection` method.

Algorithm 1 Create a Timed Automaton from a method

```
    Input: $m$, Java method that deals with time
1: function GENERATE_TA($m$)
2:     $E \leftarrow$ identify time variables and time method calls of $m$
3:     $c \leftarrow$ identify time statements based on $E$ which define a time constraint
4:     $s \leftarrow generate\_states(m)$
5:     $t \leftarrow generate\_transitions(m)$
6:     $\mathcal{A} \leftarrow generate\_automata(s, t, c)$
7:     return $\mathcal{A}$
```

Table I presents an excerpt of the analyzed Java 8 SDK methods which involve time. In this paper, we focus on the first three categories of time related methods. For the last category, it is not always possible to correctly detect whether a specific method call has or not has a specific timeout set by a preceding method call using static analysis. We also exclude methods for scheduling tasks in the future, such as provided by the `Timer` class. We plan to address these limitations in our future work.

B. Semantics of Java Time Statements

Using our categorization, we introduce the semantics of time for statements in Java programs. We start with introducing the general concepts for representing time variables and time related methods.

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 M_r^t as the set of methods that return time; M_t^t as the set of methods that contain a timeout parameter in their signature; and M_i^t as the set of methods that can potentially block the execution of a thread forever.

We use operational semantic [10] to define the time semantics of Java. A rule has some premises P that constitute the preconditions to apply the rule. Judgments J_1 and J_2 are some properties which are related to P. If the judgments hold, we can conclude Q.

$$\frac{J_1(x) \vdash y \qquad J_2(y) \vdash z}{P(x) \vdash Q(z)}$$

Given the time domain $\mathbb{T}$ defined by positive natural numbers, we define the rules T_1, T_2, and T_3 to model the time semantics of our three considered categories of methods. 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 Return Time. It is defined as:

Table I: Excerpt of time related methods extracted from the Java 8 SDK.

Return Time		**Explicit Timeout**	
Class	Method Signature	Class	Method Signature
System	nanoTime()	SSLSocket	connect(SocketAddress,int)
System	currentTimeMillis()	Timer	schedule(TimerTask,Date)
Clock	millis()	Thread	join(long)
Duration	get(TemporalUnit)	Thread	join(long,int)
Duration	getNano()	Thread	sleep(long)
Duration	getSeconds()	Thread	sleep(long,int)
Duration	toDays()	Process	waitFor(long,TimeUnit)
Duration	toHours()	FutureTask	get(long,TimeUnit)

Indefinite Wait		**Set Timeout**	
Class	Method Signature	Class	Method Signature
Object	wait()	InputStream	read()
Future	get()	InputStream	read(byte[])
FutureTask	get()	InputStream	read(byte[],int,int)
Thread	join()	MulticastSocket	receive(DatagramPacket)
Socket	connect(SocketAddress)	HttpURLConnection	connect()
SSLSocket	connect(SocketAddress)	ServerSocket	accept()
Process	waitFor()	DatagramSocket	receive(DatagramPacket)
CountDownLatch	await()	SSLServerSocket	accept()

$$T_1\colon \frac{x \in V^t \qquad m \in M_r^t}{\langle (V^t, \mathbb{T}), x = m(\_) \rangle \vdash (V^t[x \backslash val(m(\_))], \mathbb{T})}$$

Rule T_2 handles method calls containing a timeout represented by the variable t. It is defined as:

$$T_2\colon \frac{m \in M_t^t}{\langle (V^t, \mathbb{T}), x = m(\_, t, \_) \rangle \vdash (V^t, \mathbb{T} + val(t))}$$

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

$$T_3\colon \frac{m \in M_i^t}{\langle (V^t, \mathbb{T}), x = m(\_, t, \_) \rangle \vdash (V^t, \mathbb{T} + \infty)}$$

In the following section, we discuss our approach based on the definition of the semantics of time.

III. Extracting timed automata

Algorithm 1 shows the steps to construct timed automata from a given Java source code. First, we collect variables and methods calls which are time related and then, based on the collected information we identify the statements that define time constraints. The second step is presented in Section III-B and the remaining steps to extract a timed automaton in Section III-C. In this Section, we focus on the first step and how to identify Java time statements.

A. Collecting time information

The time semantics presented in the previous section allows us to create a set of rules to identify which Java statements are time related. We call the quadruplet V^t, M_r^t, M_t^t, and M_i^t *environment* and we denote it with the letter E.

Initially, V^t is empty, M_r^t, M_t^t, and M_i^t contain the fully qualified names of the Java 8 SDK methods that we manually collected and verified to be time related. The set M_r^t is extended by user specific methods that return time using the following rule R_m: if a method called $name$ with the list of parameters $pars$ and the method body S contains a return statement that references a time variable or a call to a method that returns time, it is added to the set M_r^t.

$$R_m\colon \frac{\begin{array}{c}\langle (V^t, M_r^t, M_t^t, M_i^t),\ S \rangle \vdash (V^{t\prime}, M_r^{t\prime}, M_t^t, M_i^t) \\ \exists r : return(S).(isVar(r) \wedge r \in V^{t\prime}) \vee (isCall(r) \wedge r \in M_r^{t\prime})\end{array}}{\begin{array}{c}\langle (V^t, M_r^t, M_t^t, M_i^t),\ name(pars)\ \{\ S\ \} \rangle \\ \vdash (V^t, \{name\} \cup M_r^t, M_t^t)\end{array}}$$

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.

Next, we handle the list of statements that constitute the method body. The general rule R_{stms} shows how to process such a list statement by statement.

$$R_{stms}\colon \frac{\langle E, S' \rangle \vdash E' \qquad \langle E', S'' \rangle \vdash E''}{\langle E, (S'; S'') \rangle \vdash E''}$$

Depending on the statement type, we use a different rule to process it. Due to space constraints, we present the definition of only three types of statements, namely if, while loop, and single programming statement. Moreover, we present only the definition for the while loop since all other loop statements can be easily converted to it. For the if and while loop statements defined by the rules R_{if} and R_{loop} respectively, we first process the guard B and then the list of statements inside their bodies S' and S'', and respectively S.

$$R_{if}\colon \frac{\langle E, B \rangle \vdash E_0 \qquad \langle E_0, S' \rangle \vdash E' \qquad \langle E_0, S'' \rangle \vdash E''}{\langle E, (\boldsymbol{if}\ (B)\ \{\ S'\ \}\ \boldsymbol{else}\ \{\ S''\ \}) \rangle \vdash E_0}$$

$$R_{loop}\colon \frac{\langle E, B \rangle \vdash E' \qquad \langle E', S \rangle \vdash E''}{\langle E, (\boldsymbol{while}\ (B)\ \{\ S\ \}) \rangle \vdash E''}$$

Single programming statements can further extend the environment E with variables that are time related. We consider the following three types of single Java statements:

- The variable is assigned the result of a method that returns time. This type is handled by the rule R_1.
- The variable is used as timeout parameter in a Java method call. This type is handled by the rule R_2.
- The variable is used in a mathematical expression with other time variables or time methods. This type is handled by the rule R_3.

We define $pos(\cdot)$ as the function that returns the position of the input parameter in the method call. The function $timeoutpars(\cdot)$ returns the set of indexes of the parameters that define the timeout of the input method. R_1 and R_2 are then defined as:

$$R_1\colon \frac{m \in M_r^t}{\langle (V^t, M_r^t, M_t^t, M_i^t), x = m(\_) \rangle \vdash (\{x\} \cup V^t, M_r^t, M_t^t, M_i^t)}$$

$$R_2\colon \frac{m \in M_t^t \wedge pos(y) \in timeoutpars(m)}{\langle (V^t, M_r^t, M_t^t, M_i^t), m(\_, y, \_) \rangle \vdash (\{y\} \cup V^t, M_r^t, M_t^t, M_i^t)}$$

Regarding the mathematical expressions, we consider two situations: expressions that contain only scalar values or time variables modeled by rule R_3^a and expressions that contain also calls to methods that return time modeled by rule R_3^b. R_3 is then defined as the disjunction of R_3^a and R_3^b.

$$R_3^a\colon \frac{\begin{array}{c}[(y \in V^t \wedge d \in V^t) \vee (y \in V^t \wedge d \in \mathbb{N}^+) \\ \vee (y \in \mathbb{N}^+ \wedge d \in V^t)] \wedge \odot \in \{+, -, *, /\}\end{array}}{\begin{array}{c}\langle (V^t, M_r^t, M_t^t, M_i^t), x = y \odot d \rangle \\ \vdash (\{x, y, d\} \cup V^t, M_r^t, M_t^t, M_i^t)\end{array}}$$

$$R_3^b\colon \frac{y \in M_r^t \vee d \in M_r^t \qquad \odot \in \{+, -, *, /\}}{\begin{array}{c}\langle (V^t, M_r^t, M_t^t, M_i^t), x = y \odot d \rangle \\ \vdash (\{x, y, d\} \cup V^t, M_r^t, M_t^t, M_i^t)\end{array}}$$

In the next subsection, we present the rules to determine under which condition the identified Java time statements represent a time constraint.

B. Inferring Time Constraints

In this section, we introduce a set of rules to infer two specific time related information from the source code: time assignments and time constraints. These rules constitute the second step of Algorithm 1. A time assignment is a statement that assigns a time value to a variable. A time constraint is a condition in the execution of a program that involves time. Our approach supports three types of time constraints: Explicit Timeout, Indefinite Timeout, and Time Expired. Explicit Timeout comprises calls to methods from the set M_t^t. Indefinite Timeout comprises calls to methods from the set M_i^t. Time Expired comprises comparison expressions in the Java source code that reference a time variable from V^t or a call to a method of the set M_r^t that return time. We currently support the following Java comparison operators $\otimes$:

$$\otimes ::= < \mid \leq \mid == \mid != \mid > \mid \geq$$

```
 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 = readValue();
 7        lastRefresh = System.currentTimeMillis();
 8      }
 9      public int read(){
10        long now = System.currentTimeMillis();
11        if(now - lastRefresh > MAX_TIME){
12          value = readValue();
13          lastRefresh = System.currentTimeMillis();
14        }
15        return value;
16      }
17  }
```

Listing 1: Example of a Time Expired constraint at line 11. The execution of the statements inside the if body can be performed only when enought time is passed (ten minutes).

Listing 1 presents an example of a Time Expired constraint contained by the class `Cache`. The value of the cache in the `read()` method is only updated if a certain amount of time has passed. For this, the update is guarded by an if statement in Line 11 that compares whether the amount of time passed since the last refresh is larger than the amount of time specified by the constant `MAX_TIME`.

In the following, we introduce the rules to extract the three types of time constraints from the source code of Java methods. Let E denote the environment quadruplet $(V^t, M_r^t, M_t^t, M_i^t)$. First, we parse all the classes of a Java project applying the rules defined in the previous subsection collecting all the methods that extend M_r^t. We iterate until a fix point is reached and no more methods are found. Then, we determine all class attributes that are time related. They represent the initial set V^t of time variables.

Let C_g be the set of statements which define a time constraint and C_u be the set of statements that assign a time value to a variable. Our approach starts from analyzing the method definition with rule C_m in which we first, compute all time statements and then, we apply recursively our rules to the list of statements S that constitute the method body. Rule C_{stms} shows how to process such a list of statements, collecting the time constraints.

$$C_m\colon \frac{\begin{array}{c}\langle E, S \rangle \vdash E' \\ \langle (C_g, C_u, E'), S \rangle \vdash (C_g', C_u', E')\end{array}}{\langle (C_g, C_u, E),\ name(pars)\{\ S\ \} \rangle \vdash (C_g', C_u', E')}$$

$$C_{stms}\colon \frac{\begin{array}{c}\langle (C_g, C_u, E), S' \rangle \vdash (C_g', C_u', E) \\ \langle (C_g', C_u', E), S'' \rangle \vdash (C_g'', C_u'', E)\end{array}}{\langle (C_g, C_u, E), S'; S'' \rangle \vdash (C_g'', C_u'', E)}$$

Every type of statement is processed with a different rule. Due to space constraints, we present only the rules for the following statement types: if expression, while loop, boolean expression, assignment, and method call.

If the while loop has time constraints in its guard B, they are valid only inside the while body S. The same idea holds for the if rule, in which we propagate the constraint found in the guard B inside the if body S.

$$C_{while}\colon \frac{\begin{array}{c}\langle (C_g, C_u, E), B\rangle \vdash (C_g^B, C_u^B, E)\\ \langle (C_g^B, C_u^B, E), S\rangle \vdash (C_g', C_u', E)\end{array}}{\langle (C_g, C_u, E), \textbf{\textit{while}}\ (B)\ \{\ S\ \}\rangle \vdash (C_g' \cup C_g^B, C_u', E)}$$

$$C_{if}\colon \frac{\begin{array}{c}\langle (C_g, C_u, E), B\rangle \vdash (C_g^B, C_u^B, E)\\ \langle (C_g^B, C_u^B, E), S'\rangle \vdash (C_g', C_u', E)\\ \langle (C_g, C_u', E), S''\rangle \vdash (C_g'', C_u'', E)\end{array}}{\begin{array}{c}\langle (C_g, C_u, E), \textbf{\textit{if}}\ (B)\ \{\ S'\ \}\ \textbf{\textit{else}}\ \{\ S''\ \}\rangle\\ \vdash (C_g' \cup C_g'', C_u' \cup C_u'', E)\end{array}}$$

The boolean expression rule C_{bool} searches for instances of the Time Expired constraint. We verify that at least a time variable or a call to a method of M_r^t is involved in the comparison expression.

$$C_{bool}\colon \frac{\begin{array}{c}x \in V^t \vee x \in M_r^t \vee y \in V^t \vee y \in M_r^t\\ \odot \in \{<, <=, ==, !=, >=, >\}\end{array}}{\langle (C_g, C_u, E),\ x \odot y\ \rangle \vdash (\{x \odot y\} \cup C_g, C_u, E)}$$

According to rule T_2 of our time semantics, assignments can update the value of time variables. Therefore, we check if a variable is assigned the return value of a call to a method of M_r^t or the value of another time variable. We split the check into three rules, C_{update}, C_{update}', and C_{update}''. The first rule covers the assignment with a simple method call and the second rule covers the cases in which a method call is involved in a mathematical expression. The third rule covers the cases in which the value of a time variable is assigned to another time variable with the possibility that it is involved in a mathematical expression.

$$C_{update}\colon \frac{m \in M_r^t}{\langle (C_g, C_u, E), x = m(\_)\rangle \vdash (C_g, \{x := m(\_)\} \cup C_u, E)}$$

$$C_{update}'\colon \frac{m \in M_r^t \wedge d \in N^+ \wedge \odot \in \{+, -, *, /\}}{\begin{array}{c}\langle (C_g, C_u, E),\ x = m(\_) \odot d\rangle\\ \vdash (C_g, \{x := m(\_) \odot d\} \cup C_u, E)\end{array}}$$

$$C_{update}''\colon \frac{y \in V^t \wedge d \in N^+ \wedge \odot \in \{+, -, *, /\}}{\begin{array}{c}\langle (C_g, C_u, E),\ x = y \odot d\rangle\\ \vdash (C_g, \{x := y \odot d\} \cup C_u, E)\end{array}}$$

The rule C_{call} defines how to analyze method calls identifying Explicit Timeout and Indefinite Timeout constraints. For each method call, the rule checks the environment if the call is to a method contained by M_t^t or M_i^t. If yes, the statement is added to the list of statements which identify a time constraint.

$$C_{call}\colon \frac{m \in M_t^t \vee m \in M_i^t}{\langle (C_g, C_u, E), m(\_)\rangle \vdash (\{m(\_)\} \cup C_g, C_u, E')}$$

The next Section presents the last steps of Algorithm 1 in which we extract the timed automaton states and transitions.

C. Constructing Timed Automata

We now present how we can map the constraints extracted with our approach presented in the previous section to an UPPAAL [11] timed automaton. UPPAAL is a model-checker toolbox based on the theory of timed automata. It uses a temporal logic named Timed Computation Tree Logic (TCTL) [12], [13] as a query language to describe desired properties of (networks of) timed automata. An UPPAAL timed automaton is an extension of the classic timed automaton defined by Alur and Dill [14] which has richer features, such as integer variables and channels.

Let $convert(\cdot)$ be the function that creates the automaton state given a code statement, and $stm(\cdot)$ be the function that returns the code statement associated with the given automaton state. Given a method $\mathcal{M}$ with the set of statements $\mathbb{S}$, we create the timed automaton $\mathcal{A} = \langle \Sigma, S, S_0, C, I, T\rangle$ as follows:

- $\Sigma = \emptyset$
- $S = \{convert(stm) \mid \forall stm \in \mathbb{S}\}$
- $S_0 = s$ s.t. $s \in S$ is the conversion of the first statement in $\mathbb{S}$.
- $C = \{t_0\}$ where t_0 is the variable used to track the execution time for the Explicit Timeout constraints.
- $(s, c) \in I$ iff in $stm(s)$ we found the time constraint c.
- Let $stm(s')$ be the successor of $stm(s)$ in the control flow graph of $\mathcal{M}$. T is defined as follows:
  - $(s, s, \epsilon, \emptyset, \emptyset) \in T \wedge (s, s', \epsilon, \emptyset, \emptyset) \in T$ iff $stm(s)$ has an Indefinite Timeout constraint.
  - $(s, s', \epsilon, t_0, \emptyset) \in T$ iff $stm(s')$ has an Explicit Timeout constraint and t_0 is the unique clock variable that keeps track of the execution time.
  - $(s, s', \epsilon, \emptyset, \{c\}) \in T$ iff $stm(s)$ has an Explicit Timeout or Time Expired constraint.
  - $(s, s', \epsilon, \emptyset, \emptyset) \in T$ otherwise.

The alphabet Σ is empty because we are interested only in the time behavior of the automaton. We chose to use only ϵ-transitions. The ϵ-transitions can introduce a problem with branching instructions creating a nondeterministic automaton. Since we are interested only in time properties of the code, which branch is taken in an automaton run is not important. However, branching instructions in the source code can appear only within boolean expressions which can define Time Expired constraints. If a branching instruction has a Time Expired constraint, the respective transition in the automaton is guarded with such a time constraint. In this manner, the transition can be taken only when the constraint is satisfied.

The set of states is composed of the same statements in the source code modeled as committed states in UPPAAL. A committed state enforces the execution of the automaton to always take a transition if it is enabled. In this manner, the

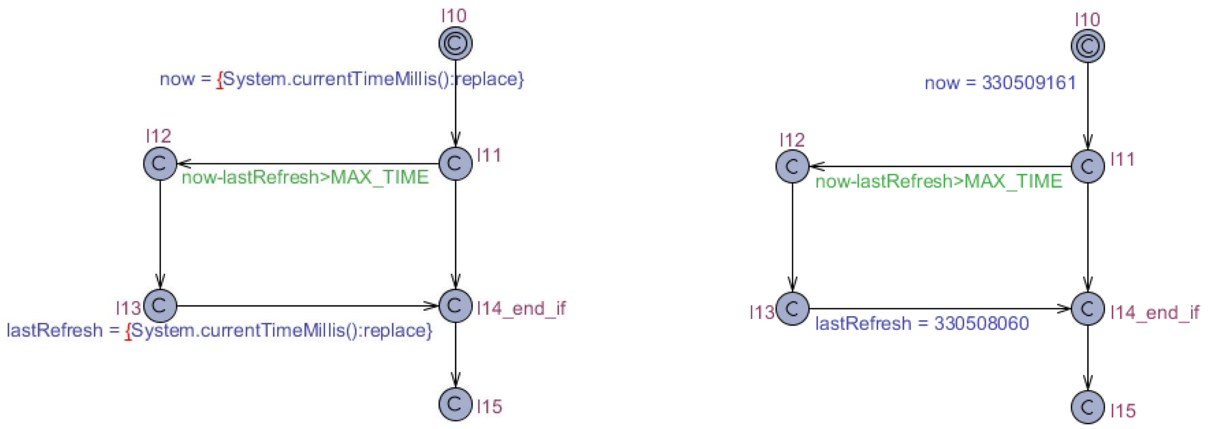

Figure 1: The graph on the left depicts the timed automaton extracted from the source code presented in Listing 1. State names represent the source code line numbers. The graph on the right depicts the same automaton with concrete values inferred from executing the program.

automaton run will simulate the execution of the control flow of the source code.

From the source code we extract time constraints and assignments to time variables. Explicit Timeout constraints are those method calls which have an upper bound limit in their execution. We model this behavior creating a unique clock variable t_0 that keeps track of the execution time. If a statement has a call to a method which defines an Explicit Timeout constraint, we reset the value of the clock variable to zero. Moreover, we enforce that at the end of the execution of the method call, the clock variable must be smaller or equal to the timeout value.

An Indefinite Timeout constraint is modeled with a transition that creates a self loop on the state that holds the method call that defines the time constraint. In this manner, the automaton can either stay in the same state or proceed to the next one.

Time Expired constraints may contain multiple variables and every distinct program variable that is assigned with a time value is translated to an integer variable in the automaton. We guard the respective transition with the same time expression found in the source code. We do not model source code time variables with clock variables in the automaton because they do not have an equal semantic. In the source code, we do not have the constraints that a clock variable has. Thus, we use the integer theory of UPPAAL to model this behavior and the statements of C_u which assign a time value to variables are modeled as transition update.

In UPPALL, a transition can have a guard and an update section. The guard contains the constraint that enables the transition and the update contains the assignment of new values to variables. When the transition is taken in a run of the automaton, the assignments in the update section are executed.

The extracted timed automata contain expressions, whose values are not know at compile time. In the following section, we present our approach to infer the values of these expressions through dynamic analysis.

D. Replacement of Expressions

We use UPPAAL to visualize and verify properties of the automata extracted from source code. However, with the approach presented in the previous section, the automata that we extract are not directly amenable to verification, since not all values of the expressions involving time can be derived by our static analysis.

The graph on the left hand side of Figure 1 shows an example of the automaton that we extract from the source code of the method `read()` displayed in Listing 1. Since the values of method calls to `System.currentMillis()` are not known at compilation time we add wildcards (between curly brackets) to mark such expressions. These wildcards need to be replace later on to use the automaton for verification. Currently, developers need to manually replace the wildcards with their values to verify the method under different scenarios.

The retrieval of such values is not always possible because we do not provide any abstraction for the information defined outside of the considered method. However, we provide a dynamic analysis approach to monitor the program and obtain proper values for the wildcards. As a first step, in the static analysis, we record variables and method calls that are referenced in statements involving time. In particular, for each expression we record the line number in the source code, the fully qualified names of the class and method containing that expression, an the expression itself. We implemented this approach as an agent that can be injected into the execution of the program. At the starting of the Java Virtual Machine, the agent loads the list of recorded expressions and rewrites the bytecode of the methods that contain those. In particular, it inserts a logging statement after each expression and before each termination point (*e.g.*, return statement) of the method. The log statements output 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 expression.

Table II: Selected Java projects used in the evaluation with number of classes (NOC), number of methods (NOM), and number of method with a method call to Java 8 SDK methods implementing time (NOMT).

Project	Version	NOC	NOM	NOMT
ActiveMQ	5.14.0	4778	36864	1121
Airavata	0.15	3988	30112	77
DeepLearning4j	0.8.1	1611	12268	84
Flume	1.8.0	1014	6491	281
Hadoop	3.0.0-alpha2	12167	92467	1816
HBase	2.0.0	8831	113226	1009
Jetty	9.3.9.v20160517	3555	22171	487
Kafka	0.10.2.0	1277	7834	91
Lens	2.6.0-beta	976	7868	84
Sling	9	5427	32385	667
Total		**43624**	**361686**	**5717**

We parse the traces created for each execution of a method, grouping them by thread id. For each thread and for each method execution, we create the respective timed automaton replacing the wildcards with the actual values. Moreover, we check if there are cycles in the execution of the method and if yes, we create an automaton for each loop iteration. Since this potentially leads to many automata for a method, we plan to investigate ways to aggregate such automata in future work. The graph on the right hand side of Figure 1 depicts an instance of a timed automaton extracted for the `read()` method with concrete values for the variables `now` and `lastRefresh`. This automaton can be used with UPPAAL to formally verify time properties of this method.

IV. Evaluation

In this section, we present the evaluation of our approach to extract Timed Automata from Java programs. For this, we implemented our approach in a prototype tool[1] that we used to perform an empirical study on 400 Java methods. With the results obtained from the empirical study, we aim to assess the precision and recall of our approach. In addition, we demonstrate how our approach can be used to detect bugs related to time by analyzing 5 real bugs obtained from the Jira issue tracker of the Apache Software Foundation.[2]

A. Experimental Setup

For the evaluation, we performed an empirical study with 10 open source Java projects listed in Table II. All of them use multi-threading and distributed components. ActiveMQ is a message broker and Airavata is a software suite to compose, manage, execute, and monitor large scale applications and workflows on computational resources. DeepLearning4J is a Java based toolkit for building, training and deploying Neural Networks. Flume is a distributed service for collecting and aggregating log data and Hadoop is a map-reduce implementation. HBase is a distributed database based on the Hadoop distributed file system. 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. Sling is a web framework that uses a Java Content Repository to store and manage content.

[1]https://github.com/rtse-project/extracting-time-automata
[2]https://issues.apache.org/jira/

Table III: Size distribution of the 400 methods used in the manual evaluation.

	Min	Q1	Mean	Q3	Max
Lines of Code	1	12	30.15	39	180
# of Statements	1	12	27.62	35	191

Table II also shows descriptive statistics computed for each project. The size of the projects varies from 976 to 12167 classes (column NOC), whereas Hadoop is the largest project. Only a small fraction of these methods contain a call to a time method of the Java 8 SDK as indicated by the numbers in column NOMT. This is expected since developers tend to wrap method calls using time with their own data structures. The project with the largest number of time method calls is Hadoop with 1816 and the project with the largest percentage of methods containing a call to a time method is ActiveMQ with roughly 3% of the methods.

Regarding the execution of the study, we randomly selected 400 from the 5717 methods which have at least one method call to a Java API time method. Our sample set exceeds the minimum number of 361 methods needed to obtain results at a 95% level of confidence with a 5% margin of error. The evaluation dataset contains methods that range from one single line of code to 180. On average the number of statements for a method is around 28 statements.

For each project, we first ran our prototype tool on the full source code to populate the values for M_r^t, as described in Section III-B. Then, we applied our approach to the source code of each selected method to extract their timed automata. In total, extracting the timed automata for the given 400 methods required 57.463 seconds. All the tests were performed on a machine with 2.5GHz Intel CPU with 16GB of physical memory. The extracted timed automata were used to compute the precision and recall of our approach as presented in the following subsection.

B. Evaluation of Precision and Recall

We evaluated the precision and recall of our approach by manually validating whether the extracted time information are also found in the source code of each method. We compute precision and recall of our implementation as:

$$Precision = 1 - \frac{NTA \text{ wrongly extracted}}{NTA \text{ extracted}}$$

$$Recall = 1 - \frac{NTA \text{ missed to extract}}{NTM \text{ extracted}}$$

We denote with NTA the number of time assignments and time constraints extracted automatically by our approach and with NTM the number of time assignments and time constraints manually extracted from the source code. Our manual evaluation of the 400 methods obtained in total 756 time constraints and time assignments compared to 723 constraints and assignments extracted by our prototype tool. Overall, the values show that our approach is able to extract timed automata with a precision of 98.62% and recall of 95.37%.

Through the manual analysis we also discovered different situations in which our approach failed to extract time information. Currently, our analysis does not consider anonymous class declarations and it does not examine the time statements inside their methods. Furthermore, our prototype tool wrongly interprets 10 string concatenation statements that use the "+" operator to combine the different values. In the implementation, these assignment statements match the rules R_3^a and R_3^b. Since the result of these statements is a string and not an integer, they should not be included in the timed automaton.

The next section presents how, through the application of our approach to big open source projects, we were able to confirm bugs which involve time.

C. Bugs in Apache Projects

In addition to the quantitative evaluation, we also performed an initial assessment of the effectiveness of our approach to detect bugs in the usage of time. For this, we manually investigated the Jira issue tracker of Java projects of the Apache Software Foundation seeking for bugs involving time. We searched the Apache issue tracker for reports that contain the keyword `timeout` applying filters to return bug reports only for Java projects reported between January 1st and May 1st, 2017. We manually filtered the results removing the bug reports that were not dealing with timing issue in the source code. The filtering was necessary because the majority of the bug reports request to adjust the timeout parameters for the integration test suite. From the remaining list of bug reports, we selected the first 5 reports, one from Flume, one from HBase, two from Kafka, and one from Lens. 4 bug reports come with an accepted patch attached to their reports. At the moment of writing, one issue, namely FLUME-3044, was still open with a proposed patch attached to it. Table IV presents the list of bugs and their short descriptions taken from the issue tracker.

The five bugs contain errors that are due to missing time constraints to guard the execution of the methods. We divided the issues into two categories:

- **Missing Upper-bound**: The method uses a loop to repeat the execution of a piece of code. The loop, however, misses a condition to guard its upper-bound execution time leading to an endless execution of the loop. The bugs KAFKA-3540, LENS-1032, and FLUME-3044 belong to this category.
- **Indefinite Wait**: The method performs a call to a method of the third category presented in Section II-A that potentially can block the execution of a thread forever. The bugs KAFKA-4306 and HBASE-17341 belong to this category.

To evaluate our approach, we applied it to both, the buggy and the patched code, extracting a timed automaton for each method modified by the bug fix. Furthermore, we instrumented the unit tests of each project to run the agent presented in Section III-D. We combined the extracted traces with the timed automaton to create several instances of the automaton.

For the bugs of the first category, we checked the transitions of the extracted automaton verifying that the loop statements does not contain a time constraint. Concerning the bugs of the second category, we used our agent to generate the different instances of the automaton. Then, we loaded these instances into UPPAAL and formally verified that each automaton can always terminate. We did this by executing the formula $A <> s_i$, where s_i is a state that allows the method to terminate correctly. The formula checks whether in every possible path the location s_i can be eventually reached from the starting state. Our findings confirm the presence of all issues and the correctness of the proposed patches.

Table IV: List of bugs taken from the Apache issue tracker. We present id, version affected, and a small summary of the issue description.

BUG ID	Version	Description
KAFKA-3540 [3]	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.
KAKFA-4306 [4]	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 [5]	2.5.0-beta	We should provide option to kill the query upon timeout for users who are not interested in result beyond timeout.
HBASE-17341 [6]	2.0.0	In *ReplicationSource.terminate()*, a Future is obtained from *ReplicationEndpoint.stop(). Future.get()* is then called, but can potentially hang there if something went wrong in the endpoint *stop()*.
FLUME-3044 [7]	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.

V. Discussion & Threats to Validity

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

[3] https://issues.apache.org/jira/browse/KAFKA-3540
[4] https://issues.apache.org/jira/browse/KAFKA-4306
[5] https://issues.apache.org/jira/browse/LENS-1032
[6] https://issues.apache.org/jira/browse/HBASE-17341
[7] https://issues.apache.org/jira/browse/FLUME-3044

A. Discussion

Based on our semantics introduced in Section II, we defined a set of rules to extract time assignments and time constraints from the source code of Java methods in Section III-A and Section III-B, respectively. From this information a timed automaton is constructed which is directly amenable to verification of time properties of the program, in our case methods. With our approach such automata can be automatically derived from source code, in contrast to existing approaches, that only model time as a sequence of events (*e.g.*, [8]) or require developers to manually extract a timed automaton (*e.g.*, [4]).

Developers can use our automata to check the time behavior of methods. For fully exploiting the formal verification power of the existing tools, such as UPPAAL, developers need to provide the values for the parameters in the automaton that could not be resolved by our approach. We simplify and support the verification process by providing an agent as part of our prototype tool implementation that can be attached to the program execution to monitor and extract concrete values for these parameters. The resulting instances of an automaton enable, for instance, the verification of safety and liveness properties. In our evaluation, we used this approach to verify that a method can always terminate correctly avoiding infinite loop executions.

Currently, there are some limitations in the extraction of timed automata from source code of a method. We do not consider calls to methods that schedule tasks in the future and methods that have a timeout set by a preceding method call. Furthermore, we currently do not consider anonymous classes and methods contained by them.

B. Threats to Validity

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

Internal Validity. The internal validity threat indicates the reliability of our prototype implementation. We mitigated this threat testing the prototype tool manually and with unit tests. For the manual analysis, we randomly selected 400 methods to evaluate the precision and recall of our approach. The size of our sample set is larger than the minimum number (361) required to obtain results at a 95% confidence level with a 5% margin of error. Moreover, we show the usability of our approach and its application on 5 bugs taken from open source Apache projects. Our approach exhibits the existence of the bugs and also validates the correctness of the proposed patches. The statistics presented in Table II show that, on average, only roughly 2% of the methods of the projects contain a call to a time method of the Java 8 SDK. Developers tend to wrap those method calls with their own data structures. To mitigate this phenomena, in our approach we collect the user defined methods which return time. However, we currently do not consider user defined methods that accept a timeout as parameter or could wait endlessly for an event. Moreover, we can address the wrapping of time method call only for the readable source code. Calls to a method defined in a third part library that wraps a time method call are not detected. Furthermore, in our analysis we do not consider a specific set of time methods found in the Java 8 SDK which are methods that change their behavior with a time constraint that is set by a preceding method call. We do not consider them because it is not always possible to correctly detect whether a specific method call has or not has a specific timeout set by a preceding method call using static analysis.

External Validity. The external validity threat concerns the generalization of the results to other software projects. We mitigated this threat by choosing ten open source Java projects of different size and of different communities to improve the generalization of our results. We also implemented our approach with a prototype tool that can be applied to other Java projects to extend our studies. Furthermore, based on our formal semantics of time, our approach can also be adapted to other programming languages, such as C#, that use a similar semantics of time as used by Java.

VI. Related Work

One of the main contribution of this paper is a semantics to determine which Java statements are time related. In the domain of semantics for the Java programming language, Bogdanas and Roşu [9] present a formal semantics for Java version 1.4 based on their $\mathbb{K}$-Framework [15]. They formalize the language syntax and how the Java Virtual Machine (JVM) interprets the bytecode. Similarly, the Real-Time for Java Expert Group [16] 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.

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. [17] present a library for UPPAAL that is able to reproduce the semantics of major Java concurrent and synchronization mechanisms. Yang et al. [18] present a tool that translates the Simulink Stateflow into UPPAAL timed automata for verification. With the verification power of UPPAAL, their approach manages to find design defects that are missed by the Simulink Design Verifier. Timed automata can also be used as basic design specification to verify properties in the program. The approach presented by Jayaraman et al. [4] takes as input a network of timed automata provided by developers. The network is used as base knowledge of the subject real-time system. They monitor the execution of the program verifying its behavior conforms to the network of timed automata provided by the developers. In an analogous way, Hakimipour et al. [5] and Georgiou et al. [6], propose a technique to automatically generate a program from timed automata. Hakimipour et al. use a timed automaton to produce an RTSJ program that is executable on both single- and multi-processor platforms. Georgiou et al. present an technique to translate timed input/output automata into distributed executable Java programs.

In the domain of verifying properties of source code, NASA developed Java Pathfinder [19], 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 language for model checking. Similarly, Henzinger et al. [20], create a framework for verifying properties of C programs for the mutex API. Bandera [21] 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 in account time. The work of Walkinshaw [8] 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 representing it as a sequence of events as they happened in the execution of a program.

VII. Conclusion

In this paper, we presented an approach to automatically reverse engineer timed automata from Java methods source code. We first introduced a definition of the semantics of time in the Java programming language and then presented the different steps of our approach to first identify Java statements related to time, second, infer time constraints, and lastly, to use this information to create the timed automata for Java methods. We implemented our approach in a prototype tool to evaluate its precision and recall to extract the time information in the source code from 400 Java methods randomly selected from 10 open source Java projects. The manual evaluation of extracted automata obtained a precision of 98.62% and recall of 95.37%. Furthermore, we presented 5 examples of real bug reports obtained from the Jira issue tracker of the Apache Software Foundation. We used our approach to verify the existence of the bugs as well as the correctness of their patches. As prerequisite of our approach, we define the time semantics of the Java language. With our defined semantics, we can verify that a program behaves correctly with respect to the time semantics. It enables the identification of a new category of problems such as identify and detect (i) code smells which involve time and (ii) time invariants required and assured by methods. Future work will be dedicated on the investigation of these topics and in improving the accuracy of our approach. For the latter, we plan to address the limitations that we found in our evaluation. Furthermore, our approach can be easily generalized to other programming languages. Currently, the base knowledge of time methods is given only for the Java language. Extending the studies to other programming languages, such as C/C#, is subject to our future work.

Acknowledgment

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

References

[1] P. Dano and A. Bollin, "Down to hades and back - experiences gained in comprehending a distributed legacy system," in *Proceedings of the International Scientific Conference on Informatics*. IEEE, 2015, pp. 85–90.

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

[3] R. Alur, "Timed automata," in *International Conference on Computer Aided Verification (CAV)*. Springer, 1999, pp. 8–22.

[4] S. Jayaraman, D. Hari, and B. Jayaraman, "Consistency of java run-time behavior with design-time specifications," in *International Conference on Contemporary Computing (IC3)*. IEEE, 2015, pp. 548–554.

[5] N. Hakimipour, P. Strooper, and A. Wellings, "Tart: Timed-automata to real-time java tool," in *International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, 2010, pp. 299–309.

[6] C. Georgiou, P. M. Musial, and C. Ploutarchou, "Tempo-toolkit: Tempo to java translation module," in *International Symposium on Network Computing and Applications (NCA)*. IEEE, 2013, pp. 235–242.

[7] D. Lo, L. Mariani, and M. Pezzè, "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)*. ACM, 2009, pp. 345–354.

[8] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2008, pp. 248–257.

[9] D. Bogdanas and G. Roşu, "K-java: A complete semantics of java," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2015, pp. 445–456.

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

[11] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1-2, pp. 134–152, 1997.

[12] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal Methods for the Design of Real-Time Systems*. Springer, 2004, pp. 200–236.

[13] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[14] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[15] G. Roşu and T. F. Serbănută, "An overview of the k semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[16] G. Bollella and J. Gosling, "The real-time specification for java," *Computer*, vol. 33, no. 6, pp. 47–54, 2000.

[17] F. Cicirelli, A. Furfaro, L. Nigro, and F. Pupo, "Modelling java concurrency: an approach and a uppaal library," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2013, pp. 1373–1380.

[18] Y. Yang, Y. Jiang, M. Gu, and J. Sun, "Verifying simulink stateflow model: timed automata approach," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 852–857.

[19] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366–381, 2000.

[20] T. A. Henzinger, G. C. Necula, R. Jhala, G. Sutre, R. Majumdar, and W. Weimer, "Temporal-safety proofs for systems code," in *International Conference on Computer Aided Verification (CAV)*. Springer, 2002, pp. 526–538.

[21] J. Hatcliff and M. Dwyer, "Using the bandera tool set to model-check properties of concurrent java software," in *International Conference on Concurrency Theory (CONCUR)*. Springer, 2001, pp. 39–58.