



Verifying temporal specifications of Java programs

Francesco Spegni¹  · Luca Spalazzi¹ · Giovanni Liva² · Martin Pinzger²  ·
Andreas Bollin²

Published online: 25 May 2020
© The Author(s) 2020

Abstract

Many Java programs encode temporal behaviors in their source code, typically mixing three features provided by the Java language: (1) pausing the execution for a limited amount of time, (2) waiting for an event that has to occur before a deadline expires, and (3) comparing timestamps. In this work, we show how to exploit modern SMT solvers together with static analysis in order to produce a network of timed automata approximating the temporal behavior of a set of Java threads. We also prove that the presented abstraction preserves the truth of MTL and ATCTL formulae, two well-known logics for expressing timed specifications. As far as we know, this is the first feasible approach enabling the user to automatically model check timed specifications of Java software directly from the source code.

Keywords Software model checking · Time-dependent behavior · Java · Timed automata · SMT · Predicate abstraction

1 Motivation

Nowadays, more and more software is being developed whose behavior depends on time and on the satisfaction of given time constraints. Consequently, the most popular programming languages provide APIs to represent and manipulate time. This obviously represents a further possible source of flaws, as shown by some recent cases such as, for example, the two vulnerabilities discovered in the Linux kernel due to timestamp overflows^{1,2} or several time-related errors discovered in Java software (Liva et al. 2018). It is, therefore, essential to have verification tools to discover such flaws, and even better if this can be done by directly analyzing the code.

¹<https://nvd.nist.gov/vuln/detail/CVE-2018-12896>

²<https://nvd.nist.gov/vuln/detail/CVE-2018-13053>

✉ Martin Pinzger
martin.pinzger@aau.at

Francesco Spegni
f.spegni@univpm.it

¹ Università Politecnica delle Marche, Ancona, Italy

² Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria

Efficient Satisfiability Modulo Theory (SMT) solvers (e.g., De Moura and Björner (2008), Dutertre (2014), Cimatti et al. (2013)) have been widely used for different forms of software verification. Some examples are symbolic execution (e.g., Rakadjiev et al. (2015), Godefroid et al. (2012), Tillmann and De Halleux (2008), and Nori et al. (2009)), static analysis (e.g., tools as OpenJML, EC/Java2, Krakatoa), model checking (e.g., Armando et al. (2009), Cordeiro et al. (2011), Cimatti and Griggio (2012), Phan et al. (2015), Kahsai et al. (2016), and Cordeiro et al. (2018)), and even model checking of timed automata (e.g., Morb e et al. (2011), Kindermann et al. (2012), and Cimatti et al. (2015)).

This work focuses on software model checking of timed specifications. This means extracting a software model that, in this case, takes into account time (expressed as timestamps, durations, and other time constraints). We express such models as timed automata (Alur et al. 1990), i.e. finite automata extended with real-valued clocks that can be reset and must satisfy given clock constraints; for these reasons they are appropriate for modeling continuous time systems, in particular real-time systems. We aim at verifying programs written in Java, one of the most popular programming languages to date. Despite the widespread use of Java and the growing importance of time-dependent behaviors, only few authors (e.g., see Liva et al. (2017), Luckow et al. (2015), Schoeberl et al. (2010), and B ogholm et al. (2008)) focused on timed automata extraction from Java programs. Furthermore, these few works studied exclusively how to extract control flow automata, i.e., automata that follow the program control flow but do not take into account the state space formed by program variables. This results in an over-approximation of the program behavior, which precludes the possibility of verifying properties over program variables in various program states and, in particular, those properties that depend on variables ranging over time values such as timestamps and durations. For this reason, most of them aim at performing best-/worst-case execution time analysis (WCET/BCET) or schedulability.

This paper tries to fill the gap described above and proposes a verification methodology based on software model checking to establish the correctness of Java programs w.r.t. specifications depending on real-valued clocks. We also show that the methodology can effectively tackle real-world Java projects and it is able to detect very subtle bugs in Apache Kafka, a distributed streaming platform,³ and Alluxio, a cloud storage abstraction library.⁴

The proposed methodology has the following innovative contributions:

- A formal semantics for timed features of Java (Section 5). In this respect, we consider the following three features: (1) pausing the execution for a limited amount of time, (2) waiting for an event that has to occur before a deadline expires, and (3) comparing timestamps.
- A set of rules used to build a *network of timed automata* from *concurrent* and *time-dependent* Java programs (Section 6). In this respect, we exploit an SMT solver.
- A proof which shows that the produced network of timed automata preserves the correctness of the original Java program w.r.t. the considered family of timed specifications (Section 7).

A previous version of this work (Spalazzi et al. 2018) showed some core components of our approach. In this work, we extend the translation rules, we prove the soundness of the produced network of timed automata, and we apply it to more software projects.

³<https://issues.apache.org/jira/browse/KAFKA-4290> accessed on 15th June 2019

⁴<https://github.com/Alluxio/alluxio> accessed on 15th June 2019

In Section 2, we draw some connections between the relevant work in the area and our methodology. Section 3 introduces some theoretical backgrounds to make the rest of the work self-contained. In Section 4, we introduce some inherent limitations of software model checking that have an impact on the design choices behind the presented verification methodology. In Section 5, we formally define a semantics for the time-dependent aspects of the Java language. In Sections 6 and 7, we present the rules for extracting timed automata from Java threads and show the soundness of the approach. In Section 8, we show the applicability of our methodology to discover bugs in a running example as well as two real-world Java projects used as workbenches for the methodology itself. In particular, we show that the methodology can be helpful to discover previously unknown bugs in the software, and that such bugs are often difficult to discover by traditional test-based approaches. For our experiments, Uppaal is used to verify the timed automata networks obtained with our tool (Larsen et al. 1997). In Section 9, we collect some concluding remarks and suggest future research directions stemming from the presented work.

2 Related work

With the term *temporal property*, and its derivatives such as temporal logic, we refer to all those properties that depend on how a system evolves over time. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are examples of temporal logics to represent these types of properties. With the term *timed temporal properties*, and derivatives such as timed temporal logic, we refer to all those properties that refer to real time and constrain the values that timestamps and durations can take. Metric Temporal Logic (MTL) and Timed Computation Tree Logic (TCTL) are examples of timed temporal logics.

Software verification techniques can be classified (Jhala and Majumdar 2009; D’silva et al. 2008) into techniques that are able to work with either a *concrete* or an *abstract* software representation.

The term *concrete* indicates that such techniques are able to represent program states exactly (Fig. 1b). This approach, even if it seems attractive, is infeasible whenever there are infinite (or very large) state spaces, as is usually the case with software in many practical situations. In order to avoid infeasibility, a trade-off between time/space complexity and completeness is required.

In particular, techniques based on *under-approximate abstraction* are used, e.g., systematic execution exploration (Godefroid 1997; Havelund and Pressburger 2000; Liva et al. 2018), a kind of dynamic analysis that is “geared towards falsification” (Jhala and Majumdar 2009). In other words, it is sound, i.e., no spurious counterexamples are generated,⁵ but incomplete, i.e., some counterexamples may not be detected (Godefroid 2004). There is a similar situation with runtime verification (Bauer et al. 2011), where a formal specification is compared with a real software execution. However, in this case, there is no extraction of a model of the software to be tested.

A different approach is represented by *over-approximate abstraction*, e.g., the state-space abstraction, where the concrete state space is partitioned into equivalence classes, such that each class is an abstract state (Ball and Rajamani 2002; Beyers et al. 2007; Corbett et al. 2000; Heizmann et al. 2013; Herber et al. 2008; Kung et al. 1994; Liva et al. 2017; Pu et al.

⁵A counterexample is an execution trace that does not satisfy a given property. A spurious counterexample is an execution trace that can be observed by the verification system but cannot be observed in the original system.

2006; Sen and Mall 2016). This kind of abstraction is “geared towards verification” (Jhala and Majumdar 2009), i.e., it is complete in finding all the counterexamples at the specified abstraction level, but it is unsound because it may produce spurious counterexamples (Clarke et al. 1994). To reduce the number of counterexamples, some sort of *Counterexample Guided Abstraction Refinement* (Clarke et al. 2000) is required, where an abstract model is iteratively and automatically refined. Several techniques for both “untimed” automata (Clarke et al. 2000) and timed automata (Wang and Jiao 2014) have been proposed.

To recap, as clearly stated by Jhala and Majumdar (2009), over-approximate abstraction can prove the correctness of a software (under the condition that the abstraction is sound w.r.t. the code) whereas under-approximate abstraction, especially systematic testing, can only conjecture it. It can only reveal the presence of bugs but not their absence (this is in line with the famous Dijkstra’s sentence (Dijkstra 1969)).

With respect to the over-approximate abstraction techniques, two classic techniques are *predicate abstraction* and *control flow abstraction*.

With *predicate abstraction*, the equivalence classes (i.e., abstract states) are created using predicates over a subset of the program variables (Fig. 1c). This means that each abstract state is denoted by a Boolean combination of these predicates that over-approximate the reachable concrete states of the program (Beyer and Wendler 2012). This abstraction computation is usually done using a Satisfiability Modulo Theories (SMT) solver (Armando et al. 2009; Ball and Rajamani 2002; Beyer et al. 2007; Beyer and Keremoglu 2011; Clarke et al. 2005; Corbett et al. 2000; Cordeiro et al. 2011; Kahsai et al. 2016; Kung et al. 1994; Sen and Mall 2016). Indeed, a large set of concrete states can be collapsed into a single abstract state denoted by the (usually small) set of predicates satisfied by such concrete states.

With *control flow abstraction*, the equivalence classes are denoted by the program locations (see Fig. 1d): there exists an abstract state for each program location, i.e., for each program statement (Heizmann et al. 2013; Herber et al. 2008; Liva et al. 2017; Pu et al. 2006). Therefore, program variables are abstracted away, and the abstract state space coincides with the set of program locations. As a consequence, the abstract state space can be computed very quickly (no SMT solvers need to be involved), at the cost that several program properties cannot be verified.

It should be noticed that these techniques, in order to be defined, must refer to the semantics of the programming language in question, not only to their syntax. For what it concerns Java, the first work defining both syntax and semantics of Java with multi-threading is by Bogdanas and Roşu (2015). In their work, the semantics of Java is defined by means of the

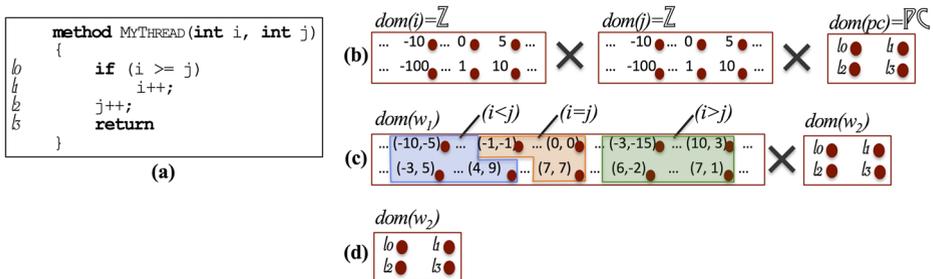


Fig. 1 An example of state space extraction: **a** its very simple source code, **b** the related concrete state space, the abstract state space based **c** on a predicate abstraction, and **d** on the control flow abstraction

\mathbb{K} -framework, a modular framework for engineering language semantics based on a set of reduction rules over configurations. A configuration is a composite and extendable algebraic structure of the program state. We extend their work introducing new information in the configurations that consider the semantic of time.

Some authors (Herber et al. 2008; Liva et al. 2017) proposed to extract control flow timed automata from a general purpose programming language, but doing this they do not take into account the role played by program variables along the execution. Therefore, these works cannot check specifications that look at the state of the program variables. Others focused on schedulability analysis and best- or worst-case execution times (Luckow et al. 2015; Thomsen et al. 2015; Schoeberl et al. 2010; Bøgholm et al. 2008), but they do not consider the correctness of the program w.r.t. properties that depend on variables ranging over timestamps and durations. To the best of our knowledge, none of them considers the problem of model checking timed properties, i.e., temporal properties of Java code that also depend on timestamp and duration variables. The methodology presented in this paper fills this gap, and we show some applications to real-world projects as an argument for its applicability and usefulness, when applied to complex software systems.

Herber et al. (2008) model-check SystemC programs, extracting timed automata from them. They also assume that most of the instructions have a zero-time model. Their approach can only handle programs containing mathematical operations over numeric variables. Our methodology overcomes this limitation, allowing to cope with user-defined data types and methods.

Blast (Beyer et al. 2007) and Ultimate LTLAutomizer (Dietsch et al. 2015) apply model checking to C programs. Blast extracts untimed control flow automata from C functions; it can only check reachability of program locations; and it only deals with Integer and Boolean program variables. Ultimate LTLAutomizer uses an SMT solver to select finite prefixes of a path and check for their infeasibility before considering the full infinite path. Therefore, it is able to verify a strict subset of liveness properties. Nevertheless, they both restrict to reachability of pre-defined error locations in the source code, and specifications cannot take into account real valued clocks and their related constraints.

Java PathFinder (JPF) (Havelund and Pressburger 2000; Cuong and Cheng 2008), its evolution Symbolic PathFinder (SPF) (Păsăreanu and Rungta 2010), and Bandera (Corbett et al. 2000) are popular model checkers for Java programs. JPF uses a Java Virtual Machine that explores symbolic paths of the Java bytecode under analysis. SPF uses constraint solvers to generate a model from Java bytecode. Bandera employs program slicing techniques for abstracting program variables that do not affect the verified specification. None of the three extracts timed automata from Java code and, therefore, they only allow the analysis of “untimed” temporal properties; i.e., according to the meaning we provided above, it is possible to check properties in LTL but not in MTL or TCTL.

More recent approaches to software verification, such as CPAChecker (Beyer and Keremoglu 2011) and SeaHorn (Gurfinkel et al. 2015), provide a modular environment where programs are manipulated through several stages to form a sort of verification “pipeline.” They differ in their internal implementation, e.g., while CPAChecker uses Control-Flow Automata as intermediate representation for the source code, SeaHorn translates the input program into Constrained Horn Clauses. Both then allow to post-process the intermediate representation of the program, in such a way that the user can select a different verification strategy for each program. Typically, the final step is to use one among several available SMT solvers in order to falsify the input (reachability) specification. Even in this case, our approach is innovative because we target real-time temporal specifications expressed in MTL or TCTL.

SymRT (Luckow et al. 2015) is a tool based on SPF that extracts networks of timed automata from Java code and is designed to verify reachability properties expressed in TCTL along with WCET/BCET and schedulability analysis. SymRT uses a control-flow abstraction as it aims at only verifying real-time properties and does not need to consider the state space produced by program variables. In our work, we use predicate abstraction and take into account program (especially time) variables and, thus, verify a wider set of specifications.

Sen and Mall (2016) apply several static analysis tools for reverse engineering a finite-state model from Java bytecode, mostly for documentation purposes. The major difference w.r.t. our approach is that they do not handle time neither in the model nor in the specification, and that they compute a transition system for each object in the program. This means they compute the state-space of each class, based on the abstract states of the class's private attributes, and compute how a method invocation can move the object from one abstract state to another. The finite-state model of the program is obtained as the combination of the transition systems of the objects used in the program itself. We are interested in employing this technique for abstracting objects *used* by threads, but we claim that it is not good enough to describe the sequence of intermediate steps taken by a thread (e.g., we may need to know in which order a thread acquires its resources in order to detect a deadlock situation).

3 Theoretical background

For the sake of self-consistency, let us collect here several formal definitions that will be used in the rest of the paper.

3.1 Networks of timed automata

Let us assume a finite set of clock variables \mathcal{C} . We call *temporal constraints* $TC(\mathcal{C})$ the terms of the grammar: $TC(\mathcal{C}) ::= \top \mid \neg TC(\mathcal{C}) \mid TC(\mathcal{C}) \vee TC(\mathcal{C}) \mid \mathcal{C} \sim \mathcal{C} \mid \mathcal{C} \sim \mathbb{T}$, where $\sim \in \{<, \leq, =, \geq, >\}$ is a comparison operator and \mathbb{T} is the time domain. In the following, we assume that the time domain is a *continuous* set (e.g., $\mathbb{T} = \mathbb{R}_{\geq 0}$). Let us call *clock valuation* any mapping $\gamma : \mathcal{C} \rightarrow \mathbb{T}$ associating clock variables to their time value in domain \mathbb{T} . Given a clock valuation γ , a time value $d \in \mathbb{T}$, and a set of clock variables $r \subseteq \mathcal{C}$,

- $\gamma + d$ denotes the clock valuation γ' such that $\gamma'(c) = \gamma(c) + d$, for every $c \in \mathcal{C}$, and
- $\gamma[r \rightarrow 0]$ denotes the clock valuation γ' where $\gamma'(c) = 0$, for every $c \in r$, and $\gamma'(c) = \gamma(c)$, for every $c \in \mathcal{C} \setminus r$.

Furthermore, let AP be finite set of atomic propositions, let M be a finite set of messages, and let $B = \{\epsilon\} \cup \{!!m, ??m : m \in M\}$ be a finite set of *broadcast labels*.

A timed automaton A is a tuple $\langle Q, \hat{q}, \mathcal{C}, \tau, I \rangle$, where $Q \subseteq 2^{AP}$ is a finite set of locations, $\hat{q} \in Q$, is a distinguished initial location, \mathcal{C} is a finite set of clock variables, $\tau \subseteq Q \times TC(\mathcal{C}) \times 2^{\mathcal{C}} \times B \times Q$ is a finite set of edges, $I : Q \rightarrow TC(\mathcal{C})$ maps locations to temporal constraints.

Let us assume the timed automata A_1, \dots, A_n , for some $n \in \mathbb{N}$, then we call *network of timed automata* the tuple (A_1, \dots, A_n) .

Intuitively, given a timed automaton $A = \langle Q, \hat{q}, \mathcal{C}, \tau, I \rangle$ with an edge $(s, \gamma, r, b, t) \in \tau$, we say that the edge is *enabled* if the current location of the automaton is $s \in Q$ and the clock variables configuration satisfy $\gamma \in TC(\mathcal{C})$. If the automaton *takes the edge* it means

(delay transition)

$$\begin{aligned}
 (\sigma, \mu) &\xrightarrow{d} (\sigma', \mu + d) \text{ if } d \in \mathbb{T}. \\
 &\forall d' \leq d. \forall i \in [1, n]. \\
 &\quad \sigma'[i] = \sigma[i] \wedge \\
 &\quad \mu'[i] = \mu[i] + d' \wedge \\
 &\quad \mu'[i] \models I_i(\sigma'[i])
 \end{aligned}$$

(discrete transition)

$$\begin{aligned}
 (\sigma, \mu) &\xrightarrow{i} (\sigma', \mu') \text{ if } i \in [1, n]. \\
 &\exists (s, \gamma, r, \epsilon, t) \in \text{enabled}_i(\sigma, \mu, \epsilon). \\
 &\quad \sigma'[i] = t \wedge \\
 &\quad \mu'[i] = (\mu[i])[r \rightarrow 0]
 \end{aligned}$$

(broadcast transition)

$$\begin{aligned}
 (\sigma, \mu) &\xrightarrow{i_1, \dots, i_p} (\sigma', \mu') \text{ if } \{i_1, \dots, i_p\} \subseteq [1, n]. \\
 &\exists m \in M. \exists (s_1, \gamma_1, r_1, !!m, t_1) \in \text{enabled}_{i_1}(\sigma, \mu, !!m). \\
 &\quad \forall k \in [1, n] \setminus \{i_1, \dots, i_p\}. \emptyset = \text{enabled}_k(\sigma, \mu, ??m) \wedge \\
 &\quad \forall j \in [2, p]. \exists (s_j, \gamma_j, r_j, ??m, t_j) \in \text{enabled}_{i_j}(\sigma, \mu, ??m). \\
 &\quad \sigma'[i_1] = t_1 \wedge \sigma'[i_j] = t_j \wedge \\
 &\quad \mu'[i_1] = (\mu[i_1])[r_1 \rightarrow 0] \wedge \mu'[i_j] = (\mu[i_j])[r_j \rightarrow 0]
 \end{aligned}$$

Fig. 2 Transitions in timed transition systems

that it updates its location to $t \in Q$ and resets all its clock variables contained in $r \subseteq C$. At any moment, if more than one edge is enabled, the system decides non-deterministically which one to take. A network of timed automaton denotes the asynchronous parallel composition of several timed automata, where each automaton keeps track of its current state and current configuration of clocks. Their execution follows the interleaving semantics, i.e., each automaton at every turn takes an enabled transition. In the following, we formally report the semantics of networks of timed automata using so-called timed transition systems.

Assume a network of timed automata (A_1, \dots, A_n) , for some $n \in \mathbb{N}$, such that every $A_i = \langle Q_i, \hat{q}_i, C_i, \tau_i, I_i \rangle$. A configuration is any tuple (σ, μ) where $\sigma[i] \in Q_i$ and $\mu[i] : C_i \rightarrow \mathbb{T}$ are a vector of locations and a vector of clock valuations, respectively. For a configuration (σ, μ) , denote with $\text{enabled}_i(\sigma, \mu, b) = \{(s, \gamma, r, b, t) \in \tau_i : \sigma[i] = s, \mu[i] \models \gamma\}$ the set of currently enabled transitions for the i th timed automaton. We write $\mu + d$ denoting the array such that $(\mu + d)[i] = \mu[i] + d$, for every $i \in [1, n]$.

Given any network of timed automata (A_1, \dots, A_n) such that $A_i = \langle Q_i, \hat{q}_i, C_i, \tau_i, I_i \rangle$, for every $i \in [1, n]$, a *timed transition system* is denoted by the tuple (S, S_0, T) where S is the set of all possible configurations, $S_0 \in S$ is the distinguished initial configuration $S_0 = (\sigma_0, \mu_0)$, where $\forall i \in [1, n]. \sigma_0[i] = \hat{q}_i$ and $\mu_0[i](c) = 0$, for all $c \in C_i$. Finally, $T \subseteq S \times S$ is the transition relation defined in Fig. 2. Note that the discrete transition is the only one moving a single timed automaton, while the others are waiting. The delay transition moves *all* the instances, increasing their clock valuations by a same amount of time d . Finally, the broadcast transition moves an instance sending the message $!!m$ together with the *maximum* set of instances capable of receiving the same message through a transition labeled with $??m$.

3.2 Real-time temporal logics

In this section, we report the formal definitions of real-time temporal logics MTL and TCTL (the interested reader may refer to (Bouyer et al. 2018) for a more complete account on the subject).

Given a set of propositions AP , the grammar for producing a MTL formula φ is the following:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{G}_I \varphi \mid \varphi \mathbf{F}_I \varphi$$

where $a \in AP$ denotes some proposition while $I \subseteq \mathbb{N} \cup \{\infty\}$ is a convex interval of natural numbers. Similarly, by restricting all time intervals I to be $[0, \infty)$, one obtains the linear-time temporal logic LTL. Missing Boolean operators (\vee, \rightarrow, \dots) and temporal operators (\mathbf{U}_I, \dots) can be defined in the usual ways.⁶

The syntax of TCTL formula φ is given by the following grammar:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbb{E}\Phi \mid \mathbb{A}\Phi$$

$$\Phi ::= a \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathbf{G}_I \varphi \mid \mathbf{F}_I \varphi$$

where, again, $a \in AP$ denotes some proposition and $I \subseteq \mathbb{N} \cup \{\infty\}$ is a convex interval of natural numbers. By restricting all intervals I to be $[0, \infty)$, one obtains the well-known branching-time temporal logic CTL.

Similarly to their untimed counterparts LTL and CTL, the two real-time temporal logics differ mainly by the semantic structure over which they are interpreted, while MTL formulae are interpreted over sets of infinite traces of state propositions, TCTL formulae are interpreted over infinite timed trees of state propositions.

We write $\rho, t \models \varphi$ to denote that the MTL formula φ holds w.r.t. time trace ρ and some point in time $t \in \mathbb{N}$. The MTL satisfiability relation \models can be defined as follows:

- $\rho, t \models a$ iff $a \in \rho(t)$, for $a \in AP$;
- $\rho, t \models \neg\varphi$ iff $\rho, t \not\models \varphi$;
- $\rho, t \models \varphi_1 \vee \varphi_2$ iff $\rho, t \models \varphi_1$ or $\rho, t \models \varphi_2$;
- $\rho, t \models \mathbf{G}_I \varphi$ iff $\rho, t' \models \varphi$, for all $t' \geq t$ such that $t' \in I$;
- $\rho, t \models \mathbf{F}_I \varphi$ iff $\rho, t' \models \varphi$, for some $t' \geq t$ such that $t' \in I$.

We write $\sigma, t \models \varphi$ to denote that the TCTL formula φ holds w.r.t. a state $\sigma \in 2^{AP}$, i.e., a subset of propositions in AP . The TCTL satisfiability relation \models can be defined as follows:

- $\sigma, t \models a$ iff $a \in \sigma$, for $a \in AP$;
- $\sigma, t \models \neg\varphi$ iff $\sigma, t \not\models \varphi$;
- $\sigma, t \models \varphi_1 \vee \varphi_2$ iff $\sigma, t \models \varphi_1$ or $\sigma, t \models \varphi_2$;
- $\sigma, t \models \mathbb{A}\Phi$ iff $\rho, t \models \Phi$, for all time traces ρ starting from σ ;
- $\sigma, t \models \mathbb{E}\Phi$ iff $\rho, t \models \Phi$, for some time trace ρ starting from σ ;
- $\rho, t \models a$ iff $a \in \rho(t)$, for $a \in AP$;
- $\rho, t \models \neg\Phi$ iff $\rho, t \not\models \Phi$;
- $\rho, t \models \Phi_1 \vee \Phi_2$ iff $\rho, t \models \Phi_1$ or $\rho, t \models \Phi_2$;
- $\rho, t \models \mathbf{G}_I \varphi$ iff $\rho(t'), t' \models \varphi$, for all $t' \geq t$ such that $t' \in I$;
- $\rho, t \models \mathbf{G}_I \varphi$ iff $\rho(t'), t' \models \varphi$, for some $t' \geq t$ such that $t' \in I$.

Since MTL and TCTL contain LTL and CTL, respectively, they inherit the property of being *not comparable*, i.e., it is neither the case that $MTL \subseteq TCTL$, nor $TCTL \subseteq MTL$.

In the following, $ATCTL$ denotes the universal segment of TCTL, i.e., the set of formulae not using the path quantifier \mathbb{E} .

⁶One exception is the “next” operator, that is usually not considered for real-time temporal logics

Running example Let us assume some shared variable y is used to count the number of processes in their critical sections. Let us assume the set of propositions $AP = \{(y \leq 1), (y > 1)\} \cup \{(thread_end, i), \neg(thread_end, i) : i \in [1, 5]\}$ describing (i) whether or not the variable y is either less than or equal to one, and (ii) whether or not the i th process reached the end of its thread, for $i \in [1, 5]$.

The usual mutual exclusion requirement, in this context, can be formalized using the following ATCTL property: $\mathbb{A}G_{\geq 0}(y \leq 1)$, meaning that at any possible moment in time, there will be at most one process in the critical section. The other common requirement, i.e., absence of starvation while waiting to enter the critical section, can be expressed with the following ATCTL formula: $\mathbb{A}F_{\geq 0} \bigwedge_{i \in [1, 5]}(thread_end, i)$.

In MTL, the mutual exclusion requirement can be expressed as $G_{\geq 0}(y \leq 1)$ while the absence of starvation can be expressed as $F_{\geq 0} \bigwedge_{i \in [1, 5]}(thread_end, i)$.

Let us observe that the presented specifications are essentially untimed since they use the operators $G_{\geq 0}$ and $F_{\geq 0}$, i.e., they demand that their sub-formulae hold at any (resp. at some) point in time, no matter how far from the begin of the execution. These properties, though, will be checked against a model that have both implicit and explicit time constraints, as explained in Section 5.7.

Let us assume some network of timed automata (A_1, \dots, A_n) and a formula $\Phi \in$ MTL. Let us write $(A_1, \dots, A_n) \models \Phi$ to denote the problem of checking whether or not the formula Φ holds in all the time traces ρ induced by (A_1, \dots, A_n) .

The model checking problem for MTL is undecidable. However, the model checking problem is EXPSPACE-complete for MITL, the subset of MTL where intervals are non-punctual (i.e., $U_{=c}$ is forbidden, for $c \in \mathbb{T}$) (Bouyer et al. 2018).

Let us assume some network of timed automata (A_1, \dots, A_n) and a formula $\Phi \in$ TCTL. Let us write $(A_1, \dots, A_n) \models \Phi$ to denote the problem of checking whether or not the formula Φ holds in the initial state of (A_1, \dots, A_n) .

The model checking problem for TCTL is PSPACE-complete (Bouyer et al. 2018).

Uppaal (Larsen et al. 1997) is a state-of-the-art model checker for networks of timed automata. It takes as input a network of timed automata and a specifications belonging to the following subset of ATCTL:

$$\Phi ::= p \mid \top \mid \neg\Phi \mid \mathbb{E}G_{\sim c}p \mid \mathbb{E}F_{\sim c}p \mid \mathbb{A}G_{\sim c}(p \rightarrow \mathbb{A}F_{\sim c}q)$$

for $p, q \in AP$, $c, c' \in \mathbb{T}$, $\sim \in \{<, \leq, =, \geq, >\}$. Note that through the usual De Morgan laws, it is also possible to verify the following universally quantified formulae as well: $\mathbb{A}G_{\sim c}p := \neg(\mathbb{E}F_{\sim c}\neg p)$ and $\mathbb{A}F_{\sim c}p := \neg(\mathbb{E}G_{\sim c}\neg p)$.

3.3 Satisfiability modulo theories

In our approach to software model checking, we make use of *satisfiability modulo theory* (SMT). This is a technique that, given a first-order logical formula, searches for a *model* of such formula, within a given set of theories. Here we report some core notions of SMT. The interested reader can find a detailed introduction on the topic in Barrett and Tinelli (2018).

Call *signature* a tuple $\Sigma = (S, P, F, \mu, \sigma)$, where S is a set of sorts, P is a set of *predicate symbols*, F is a set of *function symbols*, and $\mu : P \rightarrow S^*$ and $\sigma : F \rightarrow S^+$ are total mappings.

Each function symbol $f \in F$ specifies its *arity*, i.e., a number of accepted arguments, denoted with $arity(f) \in \mathbb{N}$, and a rank $\sigma_1 \dots \sigma_n \sigma$, denoted with $rank(f)$, where $n = arity(f)$ and $\{\sigma, \sigma_1, \dots, \sigma_n\} \subseteq S$. Similarly, each predicate symbol $p \in P$ has some arity $arity(p) = n$, for $n \geq 0$, and rank $rank(p) = \sigma_1 \dots \sigma_n$, for $\{\sigma_1, \dots, \sigma_n\} \subseteq S$.

Let us assume a signature Σ . In the following grammar, τ generates Σ -terms of sort σ while Φ generates Σ -formulae:

$$\begin{aligned} \tau &::= x \mid f(t_1, \dots, t_n) \\ \Phi &::= \perp \mid s_1 = s_2 \mid p(\tau'_1, \dots, \tau'_n) \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \exists x.\varphi_1 \end{aligned}$$

where $x \in V$ is a variable associated with some sort in S ; $f \in F$ is a function symbol such that $rank(f) = \sigma_1 \dots \sigma_n \sigma$, and terms $t_i \in \tau$ have sort σ_i , for $i \in [1, n]$; $s_1, s_2 \in \tau$ are terms of the same sort; $p \in P$ is a predicate symbol with rank $rank(p) = \sigma'_1 \dots \sigma'_m$, and $t'_i \in \tau$ is a term with sort $\sigma'_i \in S$, for $i \in [1, m]$; finally, $\varphi_1, \varphi_2 \in \Phi$.

Given a signature $\Sigma = (S, P, F, \mu, \sigma)$, a Σ -interpretation \mathcal{A} maps:

- each sort $\sigma \in S$ to a domain D_σ ;
- each variable $x \in V$ of sort $\sigma \in S$ to some element $x^{\mathcal{A}} \in D_\sigma$;
- each function $f \in F$ with $rank(f) = \sigma_1 \dots \sigma_n \sigma$ to a total mapping $f^{\mathcal{A}} : D_{\sigma_1} \times \dots \times D_{\sigma_n} \rightarrow D_\sigma$;
- each predicate $p \in P$ with $rank(p) = \sigma_1 \dots \sigma_n$ onto a relation $p^{\mathcal{A}} \subseteq D_{\sigma_1} \times \dots \times D_{\sigma_n}$.

Let us define $D = \bigcup_{\sigma \in S} D_\sigma$, i.e., the union of all the domains. Each interpretation \mathcal{A} induces a unique mapping $(_)^{\mathcal{A}} : \tau \rightarrow D$ from terms to domain elements, s.t. $(f(t_1, \dots, t_n))^{\mathcal{A}} = f^{\mathcal{A}}(t_1^{\mathcal{A}}, \dots, t_n^{\mathcal{A}})$.

Let us define a satisfiability relation between interpretation \mathcal{A} and Σ -formulae $\varphi \in \Phi$, written $\mathcal{A} \models \varphi$, by structural induction as follows:

$$\begin{aligned} \mathcal{A} &\not\models \perp \\ \mathcal{A} \models s_1 = s_2 &\iff s_1^{\mathcal{A}} = s_2^{\mathcal{A}} \\ \mathcal{A} \models p(t_1, \dots, t_n) &\iff (t_1^{\mathcal{A}}, \dots, t_n^{\mathcal{A}}) \in p^{\mathcal{A}} \\ \mathcal{A} \models \neg\varphi &\iff \mathcal{A} \not\models \varphi \\ \mathcal{A} \models \varphi_1 \vee \varphi_2 &\iff \mathcal{A} \models \varphi_1 \text{ or } \mathcal{A} \models \varphi_2 \\ \mathcal{A} \models \exists x : \sigma.\varphi &\iff \exists a \in D_\sigma. \mathcal{A}[x \mapsto a] \models \varphi \end{aligned}$$

where $\mathcal{A}[x \mapsto a]$ denotes an interpretation derived from \mathcal{A} and adding a mapping from variable x (of some sort σ) onto some term $a \in D_\sigma$.

A theory is a pair (Σ, \mathcal{M}) , where Σ is a signature while $\mathcal{M} = \{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ is a class of models sharing the same signature Σ . In this context, examples of theories typically used are the theory of equality and uninterpreted function symbols, *real* or *integer arithmetic*, *bit vectors*, and so on.

A Σ -interpretation starting from an empty set of variables is called a Σ -model. The *SMT problem* is defined as follows: taken a theory (Σ, \mathcal{M}) and a Σ -formula φ , determine whether a Σ -model $\mathcal{A} \in \mathcal{M}$ exists such that $\mathcal{A} \models \varphi$, and in case of positive answer, return it. Depending on the chosen theory, the SMT-solving problem can either be decidable or not. For instance, the theory of real arithmetic with sort \mathbb{R} , and functions symbols for *sum*, *subtraction*, and *product* is *decidable* (Enderton 1972). On the contrary, the theory of arrays with sorts A, I , and E (for *arrays*, *indices*, and *elements*, respectively) with function symbols for *read* and *write* operations is in general *undecidable* whereas its quantifier-free fragment is decidable (Bradley et al. 2006).

An *SMT solver* is a tool that, given a formula and a set of theories, returns one of the following answers: (i) a model for the formula, if it exists, i.e., an assignment of the (sorted) variables to terms of the theory; (ii) *unsat* in case such a model does not exist; (iii) *unknown*

in case a model cannot be found, but the procedure is not complete and thus cannot exclude that such a model may exist. We encode the SMT problems using the SMT-LIB v2 language (Barrett et al. 2017), a standard language for SMT solvers. For our experiments, we use the solver Z3 (De Moura and Bjørner 2008).

4 Model checking timed properties of Java programs

Software model checking has some theoretical limitations, whose knowledge is essential to establish which semantics and which extraction rules should be used and under what assumptions they can be applied.

Let us call *untimed state* (or simply state) the configuration of the variable values of a Java program. Given a Java program P and a set of states S , let us write $P \xrightarrow{?} S$ denoting the *reachability problem* asking whether the program P reaches *any* of the states in S .

Lemma 1 *Let P be a Java program with conditions, loops, and recursive types, and S any set of states. The reachability problem $P \xrightarrow{?} S$ is undecidable.*

Proof First, let us recall that the problem of detecting whether two names are aliases for the same variable is an undecidable problem for programming languages with conditions, loops, dynamic storage, and recursive data structures (Landi 1992). As Java falls under the above conditions, then the aliasing problem is also undecidable for Java. Second, such problem can be reduced to check reachability of finite state Java programs: just add a fresh variable (let us say C) initialized to 0 at the beginning, plug-in the code for which the aliasing problem should be decided (assume variable names are A and B). Then add a check like: if ($A == B$) then $C := 1$ else $C := 2$. The problem of verifying whether the program can reach a location where $C == 1$ is decidable if the aliasing problem is decidable, but the latter has been proven undecidable; thus, the reachability problem is undecidable as well. \square

The above lemma implies that, under the same assumptions, the model checking problem, for any reasonable timed or untimed temporal logic capable of expressing reachability, is undecidable as well.

One may wonder whether by restricting to Java *without recursive datatypes*, it is possible to recover decidability for the model checking problem and possibly extending it to timed formulae. The answer to this question depends on several technicalities, e.g., whether the clocks are synchronized or not among themselves, or whether we assume a dense time models vs. a discrete one, and so on.

If not otherwise specified, we assume *multi-threaded* Java programs where threads can communicate using synchronous message passing or broadcast.

We call *timed state* of a Java program the configuration of its variables together with clock variables, i.e., variables that assume values from a time domain \mathbb{T} and that are increased by some clock ticking action. A clock variable can be used, for instance, to track the execution time of a Java thread. Assume that the clock variables could be checked to enable or disable an update of the program variables and could be reset when an update of the program variables occurs. Under these assumptions, two cases can be considered: if clock variables, possibly of different Java threads, increase their internal values at the same rate, we talk about *synchronized clocks*; otherwise, we talk about *skewed clocks*.

Given n Java threads $P_1 \dots P_n$, let $P_1^{(m_1)} \parallel \dots \parallel P_n^{(m_n)}$ denote their concurrent execution, where, for each i , we have m_i instances of thread P_i .

Lemma 2 Let $P_1 \dots P_n$ be n Java (finite state) threads connected to form a clique. Let S be any set of states. Assume the use of synchronized clock variables. The reachability problem $\exists m_1, \dots, m_n. P_1^{(m_1)} \parallel \dots \parallel P_n^{(m_n)} \xrightarrow{?} S$ is decidable for timed states with only one clock variable and for a continuous time model $\mathbb{T} = \mathbb{R}$. Given a timed temporal logic formulae ϕ in MTL or TCTL, the model checking problem $\forall m_1, \dots, m_n. P_1^{(m_1)} \parallel \dots \parallel P_n^{(m_n)} \models \phi$ is undecidable.

The lemma above can be shown by reducing it to the problem of checking reachability (resp. to the recurrent state problem) in timed networks with continuous time (Abdulla and Jonsson 2003).

The next lemma, instead, shows that abstracting programs to systems with skewed clocks produce models whose model checking problem is undecidable.

Lemma 3 Let $P_1 \dots P_n$ be n Java (finite state) threads connected to form a clique. Let S be any set of states. Assume the use of skewed clock variables. The reachability problem $P_1 \parallel \dots \parallel P_n \xrightarrow{?} S$ is undecidable.

Proof The undecidability result is proven by reducing the problem of checking the reachability of a hybrid automaton with skewed clocks to the same problem on a Java program. Assume a number of Java threads equal to the number of clocks in the input automaton.

Assume an additional Java thread whose internal variables simulate the state of the input automaton.

Now, it is evident that by assuming that multi-threaded Java programs with skewed clocks can decide the reachability problem, then the same problem can be decided also for hybrid automata with skewed clocks. The latter problem, though, was proven undecidable (Henzinger et al. 1998). □

Summarizing, we have shown a few aspects of the Java language that make corresponding reachability and model checking problems undecidable.

The same analysis can indeed be replicated with minor efforts on most programming languages, since very few assumptions are made, and most programming languages satisfy them. Nevertheless, this analysis provided a motivation for drawing a formal “perimeter”

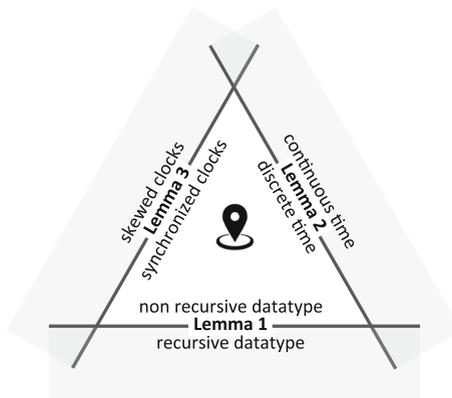


Fig. 3 Undecidability boundaries

around the kind of models that we are going to extract from Java programs. In particular, we appeal to Lemma 1 for abstracting recursive data-types to compound types. Because of Lemma 2, we prefer to assume a discrete time semantics for Java threads. Because of Lemma 3, we assume a semantics for Java where all threads increase their internal clock values at the same rate.

In Fig. 3, we depict a class of undecidable Java programs outside the triangle, as determined by the statements above. This, on the one side, justifies our design choices when giving a timed semantics for Java and, on the other side, it conveys the necessity for several abstraction techniques aiming to produce a model-checkable representation of the original program.

5 Time-dependent Java programs

Java is used for implementing software systems that have time-agnostic behaviors as well as time-dependent behaviors. By *time-dependent Java programs*, we mean Java programs containing conditional or looping statements guarded by conditions on timestamps, like the following:

```
if (now < expected_time) { do_something(); }
```

provided that `now` and `expected_time` are variables with some well-defined meaning w.r.t. the actual execution time (e.g., `now` may represent the current wall-clock time, while `expected_time` may represent a specific point in time).

The Java language does not come with a rich support of time-dependent statements and datatypes. It is also worth mentioning that the official semantics of the Java language are provided informally (Dibble et al. 2017; Bollella and Gosling 2000), while all the efforts to give a formal semantics to the Java language avoided to consider the time-related aspects of the Java language (e.g., see Bogdanas and Roşu (2015) and Farzan et al. (2004)). Even looking at several formal semantics given (mostly *a posteriori*) for other widely used programming language that we are aware of, they only describe the untimed behavior of the programming language.

This section is devoted to introduce a timed semantics of *Java*. For the sake of modularity, such semantics extends the semantics of Java 1.4 given using the \mathbb{K} -framework (Bogdanas and Roşu 2015), later referred to as KJ. The \mathbb{K} -framework, in fact, natively offers the possibility to define the semantics of programming language in a modular fashion.

The \mathbb{K} -framework allows for an operational definition of the semantics of programming languages. This is done by first defining an algebraic structure, called a *configuration*, and later a set of *rules* rewriting pieces of configurations to different pieces of configurations. A *configuration* is a set of labeled *cells*, each containing algebraic structures representing a piece of the overall current state of the program. Examples of employed algebraic structures are *lists*, *mappings*, and *stacks*. Cells may contain sets of cells as well, forming a tree-like structure. A cell written as $\langle List \rangle_{\varepsilon \circ \circ}$, for instance, has name $\varepsilon \circ \circ$ and contains a term of sort *List*. A semantic rule is represented as:

$$\text{RULE: Bar} \quad \left\langle \frac{\alpha}{\alpha'} \right\rangle_a \dots \left\langle \frac{\beta}{\beta'} \right\rangle_b \langle \gamma \rangle_c \quad \text{REQUIRES } \textit{cond}$$

where `Bar` is the (optional) rule name and several cells (e.g., `a`, `b`) may synchronously rewrite their terms (e.g., α in α' and β in β'). The term above the cell line is a *pattern* that, when it *matches* the current configuration, it rewrites the cell content with the term below

the line. A cell with no horizontal line (e.g., c) is expected to match but does not change during the rewriting. The (optional) REQUIRE clause may contain an additional condition that enables the rewriting when it holds. Note that rules can make use of variables in their matching patterns (e.g., in α, β, γ) as well as in their additional condition (e.g., in $cond$). Such variables can be referred to in the terms below the line (e.g., in α' and β') to denote the matching fragment of the configuration.

In the following, we will make use of cells with self-explanatory names: $\langle \dots \rangle_k$ contains the continuation of the evaluation of the program, $\langle \dots \rangle_{stack}$ keeps track of the stack memory of the currently executing method, while $\langle \dots \rangle_{methodContext}$ tracks the references that constitute the context of a method during its execution. The sets \mathbb{D} and \mathbb{T} represent the domain of time intervals and the domain of absolute time values, respectively. If we imagine the time as a line, then $t \in \mathbb{T}$ is used to denote a point in the time-line, while $d \in \mathbb{D}$ is used to denote the (positive or negative) displacement of two points in time.

We assume the time semantics for Java is obtained by extending the syntax of Java to allow the invocation of the following list of time-specific functions:⁷

- $future_{TT} : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{B}$: returns `true` if the first point in time is in the future w.r.t. the second one;
- $deadline_T : \mathbb{T} \rightarrow (\mathbb{T} \rightarrow \mathbb{B})$: it takes a point in time (say t) as parameter and produce a partial evaluation of the $future_{TT}$ operators, i.e., it makes t a reference time to be used for further checks against other points in time t' . Its formal definition is the following: $deadline_T(t) = \lambda t'. future_{TT}(t, t')$;
- $diff_{TT} : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{D}$: returns the displacement between any two points in time;
- $inc_{TD} : \mathbb{T} \times \mathbb{D} \rightarrow \mathbb{T}$: increases or decreases a given point in time by some given (positive or negative) duration;
- $add_{DD} : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$, $mul_{DD} : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$: adds and multiplies two given durations to obtain a third one;
- $now : \emptyset \rightarrow \mathbb{T}$: it returns some encoding of the wall-clock time;⁸
- $sleep_D : \mathbb{D} \rightarrow \emptyset$: it interrupts the execution of the computation for a given amount of time;
- $sleepUntil_T : \mathbb{T} \rightarrow \emptyset$: it interrupts the execution of the computation until a specified moment in time (if the passed time is in the past, no waiting occurs);
- $wait : \emptyset \rightarrow \emptyset$: it interrupts the execution of the computation for an unknown amount of time;⁸
- $holds_T : (\mathbb{T} \rightarrow \mathbb{B}) \times \mathbb{T} \rightarrow \mathbb{B}$: it takes a deadline as first argument and a point in time as second argument and returns whether the former is met at the specified time.

where $\mathbb{B} = \{\text{true}, \text{false}\}$ denotes the usual domain of Boolean values.

The time semantics of Java is then obtained by first extending the definition of configurations from the KJ semantics. In particular, we need to keep track of the execution time of each thread, and a set of rules for interpreting the aforementioned time-specific functions.

⁷It is well known that Java does not allow a notion of “function,” and in this presentation, we exploit exactly this fact: we are going to present rules that specify what it means to invoke such functions and such rules will not introduce any non-determinism w.r.t. the existing rules that describe the invocation of methods.

⁸The *now* operator, from a mathematically point of view, is a relation and not a function. The *wait* operator, on the other side, is better described in terms of its side effects, than in terms of its domain and codomain. In programming languages, though, it is generally accepted to call *functions* even named blocks of code that return different values for the same input data or that have side effects. We take advantage of this ambiguity for ease of presentation.

5.1 Timed configuration

Figure 4 depicts a (subset of the) timed configuration used for giving a time semantics of Java. Notice that we extended the KJ configuration by introducing new cells: $\langle Nat \rangle_{time}$, $\langle Nat \rangle_{sleep}$, and $\langle List \rangle_{deadlines}$, in each $\langle \dots \rangle_{threadData}$ cell. The aim of $time$ is to count the time units since the moment the thread was created. The cell $sleep$, instead, stores after how many time units the thread will be woken up, or -1 if the thread is not sleeping. Finally, cell $deadlines$ stores a (possibly empty) list of time values, called *deadlines*, to keep track of the moment when each of them expires and the impact on the computation execution time. In the next sections, we present the semantics only for the interesting functions. All other functions (e.g. *deadline*, *add*, ...) can be derived easily.

5.2 Rules for $sleep_D$ and $sleepUntil_T$

The semantics of functions $sleep_D : \mathbb{D} \rightarrow \emptyset$ and $sleepUntil_T : \mathbb{T} \rightarrow \emptyset$ is given by the following rules:

RULE: SleepEnter

$$\left\langle \frac{\text{functionRef}(\text{Sig})(Q) \cdot \text{RestK}}{\text{sleeping} \cdot \text{RestK}} \right\rangle_k \left\langle \frac{-1}{\text{inc}_{TD}(\mathbf{N}, Q)} \right\rangle_{\text{sleep}} \langle \mathbf{N} \rangle_{\text{time}}$$

REQUIRES Sig MATCHES $sleep_D : \mathbb{D} \rightarrow \emptyset$

RULE: SleepUntilEnter

$$\left\langle \frac{\text{functionRef}(\text{Sig})(M) \cdot \text{RestK}}{\text{sleeping} \cdot \text{RestK}} \right\rangle_k \left\langle \frac{-1}{M} \right\rangle_{\text{sleep}} \langle \mathbf{N} \rangle_{\text{time}}$$

REQUIRES Sig MATCHES $sleepUntil_T : \mathbb{T} \rightarrow \emptyset$

RULE: SleepExit

$$\left\langle \frac{\text{sleeping} \cdot \text{RestK}}{\text{RestK}} \right\rangle_k \left\langle \frac{N}{-1} \right\rangle_{\text{sleep}} \langle \mathbf{M} \rangle_{\text{time}} \quad \text{REQUIRES } future_{TT}(\mathbf{N}, \mathbf{M})$$

where M, N match two time values, and Q matches a duration value.

Here $\text{functionRef}(\text{Sig})$ means that a variable Sig matches the signature of a function (more precisely, of a method), while RestK is a variable introduced for matching the rest of the continuation to be saved for later use. Rule SleepEnter enters the sleeping state and sets the timer for exiting the sleep state to a point in time computed by the sum of the actual time and the passed duration. Rule SleepUntilEnter , on the other side, sets the timer directly to the passed point in time, independently from the actual time. Note

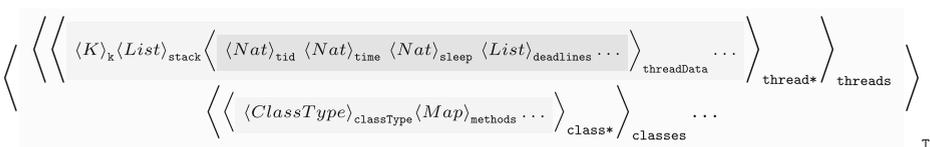


Fig. 4 Subset of configuration

5.5 Rules for time ticking

In order to describe the elapsing of time, we provide a rule named `Tick` which updates the execution time in every thread configuration. The rule is defined as follows:

$$\text{RULE: Tick} \quad \left\langle \frac{\text{Threads}}{\text{timeinc(Threads)}} \right\rangle_{\text{threads}} \quad \text{REQUIRES invHolds(Threads)}$$

where the term `Threads` matches zero or more $\langle \dots \rangle_{\text{thread}}$ cells. Auxiliary operators `timeinc` and `invHolds` are defined by means of the following equations:

$$\begin{aligned} \text{timeinc}(\langle \langle N \rangle_{\text{time}} \dots \rangle_{\text{thread}} \text{Threads}) &= \langle \langle \text{inc}_{TD}(N, 1) \rangle_{\text{time}} \dots \rangle_{\text{threads}} \text{timeinc(Threads)} \\ \text{timeinc}(\epsilon) &= \epsilon \end{aligned}$$

$$\text{invHolds(Threads)} = \begin{cases} \text{true} & \text{if Threads} = \epsilon \\ \text{invHolds}(\text{tail(Threads)}) & \text{if head(Threads)} = \\ & \langle \langle N \rangle_{\text{time}} \langle M \rangle_{\text{sleep}} \dots \rangle_{\text{thread}} \\ & \text{and future}_{TT}(M, N) \\ \text{false} & \text{otherwise} \end{cases}$$

where `N`, `M` match two time values, `Threads` matches a list of $\langle \dots \rangle_{\text{thread}}$ cells and ϵ matches an empty list.

The reasoning of the `Tick` rule is that time can advance provided that no thread will remain “asleep” after its $\langle \dots \rangle_{\text{time}}$ timer overcomes its $\langle \dots \rangle_{\text{sleep}}$ timestamp. Notice that the tick rule invariant does not prevent deadline timers to “expire”, w.r.t. the current time.

5.6 Comparison with real-time software systems

The class of time-dependent Java programs should be considered a superset of the real-time Java programs. The latter, indeed, is usually defined through combinations of several terminating tasks, each one having well-defined deadlines. The correctness of a real-time software is the result of two factors: (i) each task meets some logical requirements, and (ii) it completes its tasks in time w.r.t. some given deadlines. Deadlines of real-time tasks are usually set statically, at compile-time, and not computed at run-time. Since the actual execution time is a fundamental aspect of real-time tasks, and the task deadlines may be expressed in the scale of milliseconds, real-time tasks are executed using specially designed schedulers which give priority to tasks whose execution time is closer to their deadlines. In the case of Java real-time software, special JVMs can be used (often called *real-time JVMs*) that guarantee predictable upper bounds for every instruction of the Java language (the reader can refer to Laplante and Ovaska (2011), Hunt and et al. (2017), and Bollella and Gosling (2000) for a survey on the topic).

On the other side, we call *time-dependent program* any software that makes use of deadlines, i.e., fixed points in time used as comparisons against timing of events. In this context, deadlines *may* be computed at run-time, and the deadlines are not so tight to require a full-fledged real-time JVM for the execution of the code. The code, though, contains comparisons between time values and deadlines, and their evaluation is expected to lead the software to behave differently, in some meaningful way. In other words, the correctness

of the software is expected to depend upon the correct handling of the timing of events. Real-time deadlines can be encoded using our notion of deadlines and time operators, as follows:

```
MyTask t = MyTask(par1, ..., parN);
assertTrue(t instanceof java.lang.Thread);

// create a deadline max units from now
Deadline dl = t.setDeadline(max);

t.run();

do {
    if (! dl.holds(now())) {
        t.interrupt();
        throw new TimeoutException();
    }
} while (t.isAlive());
```

Similarly, our time-specific functions *now*, *sleepUntil_T*, and *inc_{TD}* can be used to encode a periodic task, i.e., to ensure that a piece of code is executed every *d* time units, for some integer *d* > 0:

```
MyTask t = MyTask(par1, ..., parN);
assertTrue(t instanceof java.lang.Thread);

while (! t.isFinished()) {
    if (t.isAlive()) {
        // it took more than the period to complete
        throw new TimeoutException();
    } else {
        t.run();
        awake = inc(now(), period);
        sleepUntil(awake); // stop current thread, but not t
    }
}
```

5.7 A running example

In the following, we introduce as running example the Fischer's algorithm for mutual exclusion, as presented in a classic paper by Lamport (1987).

The algorithm is designed to ensure mutual exclusion when a set of processes, running on a multi-processor system, gains access to a critical section. The core idea of the algorithm is that every process, in order to enter the critical section, must *announce* its intention by writing its own identifier in a shared variable, i.e., accessible to every processor. Then, every processor waits some amount of time, and if at the end of the waiting its name in the shared

```

        public void enterCS() throws InterruptedException {
0           do {
0.0             while (x != null) { // await
0.0.0               System.out.println("wait");
0.0.1             }
0.1             x = this.id; // announce
0.2             this.sleep(DELTA);
0.3           } while (! this.id.equals(x));
              // begin critical section
1             y++;
2             y--;
              // end critical section
3             x = null;
4           }

```

Fig. 5 A Java implementation of the Fischer's algorithm for mutual exclusion

variable has not been overwritten, it assumes that it is the only one accessing the critical section, and so it *enters* it. A Java implementation of the Fischer's algorithm is given in Fig. 5. On the left-hand side of the code, we report the encoding of the line-of-code (LOC) of each instruction, for future reference. For convenience, we represent the LOC as a stack of numbers, and we refer to Section 6 for the technical details about this encoding.

The peculiarity of the algorithm is that it uses time assumptions in place of the usual *test-and-set* operation implemented in hardware. More specifically, it assumes that any process can execute in sequence the `//await` (line 0.0) and the `//announce` (line 0.1) steps in less than `DELTA` time units. Note that `x` is the shared variable used to announce the willingness of a process to enter the critical section, while `this.id` is a local variable that stores the identity of the process itself. We declared `this.id` to be of type `String` in order to show, later, how our approach can cope with more complex data-types than numeric and Boolean types. Variable `y` is an auxiliary shared variable used to count the number of processes in the critical section. In Section 3.2, we have shown how to encode the mutual exclusion requirement as well as the absence of starvation, using real-time temporal logics.

The presented algorithm has both explicit and implicit time constraints. We call *explicit* those time constraints that are derived from a careful analysis of the source code, while we call all the other time constraints *implicit*. An example of the latter are the assumptions on the value of constant `DELTA` w.r.t. the execution time of the processes. Explicit time constraints are inferable from the usage of timestamps and the invocation of time-related methods (e.g., the variable `DELTA` and the method `sleep`). The fact that the Fischer's algorithm has both kinds of time constraints makes it a good benchmark for our methodology.

6 Abstraction rules

We begin by introducing rules that encode an (untimed) existential abstraction of Java threads. Such abstraction produces a finite-state transition system that will be labeled, later in this section, with time information to produce a network of timed automata. As we will see in more details later, the produced abstraction of the code will be *untimed* because, at this stage, timestamp variables will be treated as regular integer variables, and it will be *existential* because it will be the result of solving satisfiability problems of existentially quantified logical formulas over the program variables.

6.1 Abstracting time-independent steps

We assume a finite set of *concrete variables* $V = \{v_1, \dots, v_n\}$ and we define the *concrete state space* as $SS(V) = dom(v_1) \times \dots \times dom(v_n)$, where dom maps a variable to a set, and we call $dom(v)$ the *domain* of variable v .

We assume also a finite set of *abstract variables* $W = \{w_1, \dots, w_m\}$ and we define the *abstract state space* as $SS(W)$. We call *concrete state* and *abstract state* any item $s \in SS(V)$ and $\hat{s} \in SS(W)$, respectively.

Let us call *abstraction function* any mapping $\alpha_i : SS(V) \rightarrow dom(w_i)$. A set of abstraction functions $\alpha_1, \dots, \alpha_m$ induces an *abstraction*, i.e., a mapping $\alpha : SS(V) \rightarrow SS(W)$ such that:

$$\alpha(v_1, \dots, v_n) = (\alpha_1(v_1, \dots, v_n), \dots, \alpha_m(v_1, \dots, v_n)).$$

Running example Call P the implementation of the Fischer’s algorithm reported in Fig. 5. The thread has the variables $V = \{id, x, y, pc\}$, where pc is the register storing the thread program counter, a special purpose variable used to track the currently executed LOC. Assume that we want to abstract the thread using the following abstraction functions:

$$\alpha_1(id, x, y, pc) = \begin{cases} 0 & \text{if } id = "fie" \\ 1 & \text{if } id = "foo" \\ 2 & \text{otherwise} \end{cases} \quad \alpha_3(id, x, y, pc) = \begin{cases} 0 & \text{if } y = 0 \\ 1 & \text{if } y = 1 \\ 2 & \text{if } y > 1 \end{cases}$$

$$\alpha_2(id, x, y, pc) = \begin{cases} 0 & \text{if } x = null \\ 1 & \text{if } x = "fie" \\ 2 & \text{if } x = "foo" \\ 3 & \text{otherwise} \end{cases} \quad \alpha_4(id, x, y, pc) = pc$$

While defining the abstraction functions, we are assuming a scenario where a process identity can either be a literal between "foo" and "fie" or anything else. To this aim, the abstraction function α_1 (resp. α_2) compares the values of the local variable id (resp. of the global variable x) with the allowed identifiers. Note that since every thread is assumed to have an identifier id , there is no need for checking whether id equals $null$. Abstraction function α_3 abstracts the number of threads in their critical section, counted by the global variable y , while α_4 traces exactly the flow of the code, i.e., each change in the LOC through the special variable pc . Please note that, at this stage, while defining the abstraction of a thread, the abstraction functions are not aware of how many such threads will be in the system. Thus, the framework does not allow to specify constraints that involve local variables of different threads, neither does it need to specify that at any given time all the threads see the same value for a global variable. These aspects, on the other side, are fundamental for carrying on the verification task, because, for instance, one should demand that no two threads share the same identifier in the system. Such details will be presented in Section 8.

Let us call *thread* the state transition system $P = (S, S_0, T)$ where $S = SS(V)$ is the set of configurations that the thread variables can assume, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation between thread states induced by the thread code.

Given a thread P , let us call an *abstraction of P*, the transition system $\hat{P} = (\hat{S}, \hat{S}_0, \hat{T})$ where $\hat{S} = SS(W)$, $\hat{S}_0 = \{\alpha(s_0) \mid s_0 \in S_0\}$, $\hat{T} \subseteq \hat{S} \times \hat{S}$, such that $\hat{T} = \{(\alpha(s), \alpha(t)) \mid (s, t) \in T\}$.

We use an SMT solver in order to compute the abstract state space $SS(W)$ of the thread under analysis. In order to do so, we need to know a set of abstract variables W and

abstraction functions $\alpha_1, \dots, \alpha_m$.⁹ The questions passed to the SMT solver, in our case, are first-order logical conjunctions describing (i) some predicates holding on variables V *before* executing statement ι ; (ii) the same predicates holding on variables V *after* executing ι ; (iii) the relation induced by ι between the initial and the final values of each variable. The ability of the SMT solver to decide the received problems obviously depends on whether the predicates used to build the abstraction function fall into a decidable theory.

The finite-state automaton obtained at the end of the abstraction process is said to be a *predicate abstraction* of the original code exactly because the problem of abstracting an entire piece of code is reduced to deciding a set of logical predicates over the program variables. The details of how this process is defined are explained in the following.

Building the abstract thread \hat{P} from a concrete thread $P = (S, S_0, T)$ is quite trivial. Unfortunately, explicitly representing a concrete thread P would be an infeasible task, when not impossible (e.g., if the thread variables have unbounded domains). One of the main goals of this work is to show how to build an abstract thread \hat{P} directly from the thread source code, avoiding the intermediate step of enumerating the states and transitions of concrete thread P . We can rely, instead, on the *thread code*, i.e., the set of its Java statements and classes, and the initial state, given by assigning to each variable in the code its initial value.

From now on, assume that the set of concrete (resp. abstract) variables V (resp. W) contain the special variable pc tracking the current LOC executed by the thread, assuming such variable has domain $dom(pc) = PC$, i.e., the set of all possible locations.

In order to take into account nested statements, let us assume that PC is a dotted-separated stack of natural numbers \mathbb{N} and special symbols Σ , pushing and popping on the rightmost position.

We assume every natural number is also a member of PC , i.e., $\mathbb{N} \subseteq PC$, since every $n \in \mathbb{N}$ can be interpreted as the stack containing only n on its top. We assume the following operators over LOCs, *inc*, *push*, *pop*, defined as follows:

$$\begin{aligned} inc(pc.n) &= pc.(n + 1) & push(pc, n) &= pc.n \\ inc(n) &= n + 1 & pop(pc.\sigma) &= pc \end{aligned}$$

for every $pc \in PC$, $n \in \mathbb{N}$, $\sigma \in \mathbb{N} \cup \Sigma$. For the sake of readability, given any $pc \in PC$ and $\sigma \in \mathbb{N} \cup \Sigma$, we may write $pc.inc()$ (resp. $pc.push(\sigma)$, resp. $pc.pop()$) in place of $inc(pc)$ (resp. $push(pc, n)$, resp. $pop(pc)$).

The reason for using such data structure to model the LOC to be executed is that it reflects precisely the nested structure of the source code in structured programming languages. Thus, given a statement $stmt$ and its LOC pc , it is easy to compute the next possible LOC where the thread can jump by executing such statement. If $stmt$ is a variable assignment, the next LOC is $pc.inc()$. If $stmt$ is an if-then-else block, then the body of the “then” branch begins at position $pc.push(THEN).push(0)$, while the “else” branch begins at position $pc.push(ELSE).push(0)$; if $stmt$ is a while statement, there is only one possible body, beginning at position $pc.push(0)$; and similarly for the other Java control structures.

Since Java is a deterministic programming language, each statement at a given LOC can only jump to a new single LOC, depending on the state of the thread. Assuming an asynchronous thread semantics, a program with n threads has up to n successor states from

⁹For convenience, at the moment, we assume that the user provides such information to the methodology. It is easy to conceive that some of it can be inferred from a further step of static analysis of the source code, mixed with heuristic rules.

any given state, since in general, the choice of the next thread to run is the only form of non-determinism in the Java language specification (Dibble et al. 2017).

In the following, for a state s , we may write $s.x$ to denote the value of variable x in s . We may also write $s[x \leftarrow z]$ to denote the (unique) state obtained from s replacing the current value of x with z , provided that $z \in \text{dom}(x)$. By definition, for any $x \neq y$, we have to check that $(s[x \leftarrow y]).y = s.y$ while $(s[x \leftarrow z]).x = z$. Given any state s and symbol $\sigma \in \mathbb{N} \cup \Sigma$, we will write $s.\text{inc}()$ (resp. $s.\text{push}(\sigma)$, resp. $s.\text{pop}()$) as shorthand for state $s[pc \leftarrow \text{inc}(s.pc)]$ (resp. $s[pc \leftarrow \text{push}(s.pc, m)]$, resp. $s[pc \leftarrow \text{pop}(s.pc)]$).

Given an abstraction α and symbol $\sigma \in \mathbb{N} \cup \Sigma$, in the following, we will write $SS(\alpha, \sigma)$ to denote the set $\{s : W = \text{dom}(\alpha), s \in SS(W), s.pc = \sigma\}$. Basically, $SS(\alpha, \sigma)$ filters the abstract state space $SS(W)$ by taking only those states where program counter equals σ . Since the definition of a (abstract or concrete) state is reduced to checking a finite number of first-order predicates over the program variables, we will write $\text{predicate}(s)$ to denote the first-order predicate corresponding to state s .

In the following, we assume an SMT solver is invoked through the special function ISSAT, taking as input a first-order Boolean formula over several possible theories (typically equality, arithmetic, recursive data structures, ...). The output of ISSAT is either `true`, if a variable assignment exists that satisfies the given formula, or `false` otherwise. The ISSAT operator can be seen as a decidable or semi-decidable oracle, depending on the theory in which the input Boolean formula is expressed. Let us assume that, for any Java assignment instruction ι , we are able to compute $\llbracket \iota \rrbracket_{\text{SMT}}$, a first-order formula describing the effects of instruction ι on a given abstract state. Let us call *simple Java assignments* those assignments that have, in their right-hand side, either a single method call or an arithmetic or logical expression. Any complex Java assignment that mixes method calls with expressions in its right-hand side can be pre-processed to be replaced by a sequence of simple Java assignments with the same behavior. This step may require to introduce a finite number of auxiliary variables and it can be done using standard techniques. Thus, in the following, we assume that the Java threads under investigation have been previously pre-processed, if needed, and contain only simple Java statements.

Next, given two abstract states s and t and a Java assignment instruction ι , we say that state t is reachable from s via the instruction ι iff $\text{ISSAT}(\text{predicate}(s) \wedge \llbracket \iota \rrbracket_{\text{SMT}} \wedge \text{indexed}(\text{predicate}(t)))$, where $\text{indexed}(p)$ returns a copy of the predicate p where every variable v is replaced by an indexed copy of itself v_1 . In this case, we add a transition $s \rightarrow t$ to the discrete model of the thread under analysis. The problem is thus how to define $\llbracket \cdot \rrbracket_{\text{SMT}}$ and how the latter relates every variable v (taken from $\text{predicate}(s)$) to its indexed copy v_1 (introduced by $\text{indexed}(\text{predicate}(t))$).

Running example Let us represent a state as the tuple of values assumed by the abstraction functions $\alpha_1, \dots, \alpha_4$ defined earlier. For instance, $s = (0, 0, 0, 0.1)$ is the state where $\alpha_1 = 0, \alpha_2 = 0, \alpha_3 = 0, \alpha_4 = 0.1$. In this case, $\text{predicate}(s)$ would return the following first-order predicate: $\text{id} = \text{"fie"} \wedge x = \text{null} \wedge y = 0 \wedge pc = 0.1$, while $\text{indexed}(\text{predicate}(s))$ would return the indexed version of the same predicate, i.e., $\text{id}_1 = \text{"fie"} \wedge x_1 = \text{null} \wedge y_1 = 0 \wedge pc_1 = 0.1$.

Let us assume the states $t = (0, 1, 0, 0.2)$, i.e., $\text{predicate}(t) = \text{id} = \text{"fie"} \wedge x = \text{"fie"} \wedge y = 0 \wedge pc = 0.2$, and $u = (0, 2, 0, 0.2)$, i.e., $\text{predicate}(u) = \text{id} = \text{"fie"} \wedge x = \text{"foo"} \wedge y = 0 \wedge pc = 0.2$. Let us now check whether through statement $x = \text{this.id}$ in line 0.1, the program can reach states t and u from s , both in line 0.2. First, we need to compute the first-order logic interpretation of the statement.

The latter, given the abstraction, is: $\llbracket x = \text{this.id} \rrbracket_{\text{SMT}} = (\text{strval}(x.l) = \text{strval}(\text{id})) \wedge (\text{id}.l = \text{id}) \wedge (y.l = y)$, i.e., the instruction updates the value of string pointed by program variable x to equal the value of string pointed by id , and leaves all the other variables untouched (i.e., the indexed version of each other variable equals the corresponding unindexed variable). Next, we submit the following two satisfiability problems to the SMT solver:

- A) $\text{IS SAT}(\text{predicate}(s) \wedge \llbracket x = \text{this.id} \rrbracket_{\text{SMT}} \wedge \text{indexed}(\text{predicate}(t)))$
 B) $\text{IS SAT}(\text{predicate}(s) \wedge \llbracket x = \text{this.id} \rrbracket_{\text{SMT}} \wedge \text{indexed}(\text{predicate}(u)))$

Note that the major difference between the two SMT problems is that: problem A) has positive answer if, and only if, variable x in the program can assume value “fie” after executing the assignment statement in configuration s ; problem B), on the contrary, has positive answer if, and only if, variable x in the program can be evaluated to “foo” after the same assignment in the same configuration s . In line with the Java semantics of the assignment statement, given the configuration s only problem A) has positive answer. Thus we add the transition $s \xrightarrow{x=\text{this.id}} t$ to the set of transitions in the abstract model, while we do not add transition $s \xrightarrow{x=\text{this.id}} u$.

Let us emphasize that `strval`, appearing in our example, is a user-defined SMT function describing the interpretation of the `String.equals` method from the Java library.

Indeed, there are Java data types and operations that do not have a straightforward mapping onto the data types and operations supported by the SMT solver. We postpone to Section 6.2 a more detailed discussion about how the user can provide an interpretation for such data types and operations. Finally, note that the SMT problem does not require the value of LOC before and after the current Java instruction since it does not affect the satisfiability of the SMT problem itself; thus, the variable pc does not appear in the argument of IS SAT. The pc variable is tracked separately, to model the control-flow of the program.

Given an abstract state s and an instruction ι , we can compute the set of outgoing transitions from s when applying ι . We do this by means of several operators, one for each syntactic category of statements and expressions allowed by Java.

In Algorithms 2–8, we report the pseudocode of the operators that cover the core control structures of the Java language, viz. sequences of statements, *if-then-else*, *while* loops, *method invocations*, numerical, and logical expressions. Furthermore, let *ReachHandler* be a mapping that associates each syntactic category to a function (e.g., *if-then-else* statements are associated with REACHITE and sequences of statements are associated with REACHSEQ). Each function returned by *ReachHandler* takes the current statement $stmt$, a source state s , and an abstraction α . The returned value is a pair whose former element is the set of states reachable from the source state with the passed instruction, while the latter is the set of transitions in between. The auxiliary function ADDREACTEDGES (see Algorithm 1) enriches the passed set of states S and transitions T , allowing to react to changes of the global environment through special broadcast receiving transitions. The reason behind these transitions will be clarified later, when describing the function REACHTHREAD (see Algorithm 8), which in turn adds in the network of timed automata special broadcast sending transitions.

Algorithm 1 Auxiliary function to add special broadcast receiving transitions.

```

function ADDREACTEDGES( $S, T$ )
   $T' := T$ 
   $S' := S$ 
  for  $s \in S$  do
    for  $w$  is an abstract global variable do
      for  $a \in \text{dom}(w)$  do
         $t := s[w \leftarrow a]$ 
        if  $s \neq t$  then
           $T' := T' \cup \{s \xrightarrow{??w_a} t\}$ 
           $S' := S' \cup \{t\}$ 
  return  $S', T'$ 

```

Algorithm 2 Operator for the *sequence* of statements.

```

function REACHSEQ( $stmts, s, \alpha$ )
   $T := \emptyset$ 
   $S := \{s\}$ 
  for  $I$  in  $stmts$  do
     $S, T := \text{ADDREACTEDGES}(S, T)$ 
    for  $s$  in  $S$  do
       $handler := \text{ReachHandler}[\text{type}(I)]$ 
       $S', T' := handler(I, s, \alpha)$ 
       $S := S \cup S'$ 
       $T := T \cup T'$ 
   $S := S'$ 
   $S, T := \text{ADDREACTEDGES}(S, T)$ 
  return  $S, T$ 

```

Algorithm 3 Operator for the *assignment* statement.

```

function REACHASSIGN( $stmt, s, \alpha$ )
   $S := \emptyset$ 
   $T := \emptyset$ 
  for  $t$  in  $SS(\alpha, pc(s).inc())$  do
     $jump := \text{ISAT}(\text{pred}(s) \wedge \text{pred}(stmt) \wedge \text{pred}(t))$ 
    if  $jump$  then
       $S := S \cup \{t\}$ 
       $T := T \cup \{s \rightarrow t\}$ 
  return  $S, T$ 

```

Algorithm 4 Operator for the *if-then-else* statement.

```

function REACHITE(stmt, s,  $\alpha$ )
  jump_then := ISSAT(predicate(s)  $\wedge$  guard(stmt))
  jump_else := ISSAT(predicate(s)  $\wedge$   $\neg$ guard(stmt))
  if jump_then then
     $s' := s.push(THEN).push(0)$  ▷ enter the ‘then’ body
    nextStmt := thenBody(stmt)
    handler := ReachHandler[type(nextStmt)]
     $S', T' := handler(nextStmt, s', \alpha)$ 
     $T' := T' \cup \{s \rightarrow s'\}$ 
  if jump_else then
     $s'' := s.push(ELSE).push(0)$  ▷ enter the ‘else’ body
    nextStmt := elseBody(stmt)
    handler := ReachHandler[type(nextStmt)]
     $S'', T'' := handler(nextStmt, s'', \alpha)$ 
     $T'' := T'' \cup \{s \rightarrow s''\}$ 
   $T := T' \cup T''$ 
   $S := \emptyset$ 
  for s in  $S' \cup S''$  do
     $s' := s.pop().pop().inc()$  ▷ exit the ‘then’/‘else’ body, go to next LOC
     $S := S \cup \{s'\}$ 
     $T := T \cup \{s \rightarrow s'\}$ 
  return  $S, T$ 

```

Algorithm 5 Operator for the *while* statement.

```

function REACHWHILE(stmt, s,  $\alpha$ )
  jump_while := ISSAT(predicate(s)  $\wedge$  guard(stmt))
  skip_while := ISSAT(predicate(s)  $\wedge$   $\neg$ guard(stmt))
   $S := \emptyset$ 
   $T := \emptyset$ 
  if jump_while then
     $s' := s.push(0)$  ▷ enter the ‘while’ body
    nextStmt := whileBody(stmt)
    handler := ReachHandler[type(nextStmt)]
     $S', T' := handler(nextStmt, s', \alpha)$ 
     $T := T' \cup \{s \rightarrow s'\}$ 
    for s in  $S'$  do
       $s' := s.pop().inc()$  ▷ exit the ‘while’ body, go to next LOC
       $S := S \cup \{s'\}$ 
       $T := T \cup \{s \rightarrow s'\}$ 
  if skip_while then
     $s' := s.inc()$  ▷ go to next LOC
     $S := S \cup \{s'\}$ 
     $T := T \cup \{s \rightarrow s'\}$ 
   $S := \emptyset$ 
   $T := T'$ 
  return  $S, T$ 

```

Notice that the REACHITE operator (see Algorithm 4) allows in principle, from the same configuration, to reach some states in the *then*-branch as well as in the *else*-branch. This is consistent with the existential nature of the abstraction. Notice also that the guard g may contain statements with side effects. We address this by assuming a straightforward pre-processing at the parsing stage, rewriting the *if-then-else* statement to first decompose the complex guard g to a sequence of (intermediate) variable assignments and methods calls, and next replace g with a (functionally) equivalent guard g' without side-effects.

Operator REACHWHILE (see Algorithm 5) abstracts a loop in the code. A key step is the analysis of the loop guard. First, we build a logical formula intersecting the source state with the guard of the while loop ($predicate(s) \wedge guard(stmt)$) and if it is satisfiable it unrolls and builds the abstraction of the while body, starting from (abstract) state s . Next, a second logical formula intersects the source state with the *negation* of the loop guard ($predicate(s) \wedge \neg guard(stmt)$) and, again, if it is satisfiable, a transition is added towards the first LOC outside the while loop (i.e., $s.inc()$). Notice that, like for the REACHITE case, due to the existential nature of the abstraction, it is possible that from the same (abstract) state s , the finite-state automaton may either enter the while loop or skip it, non-deterministically.

Notice also that the loop unrolling of REACHWHILE (see Algorithm 5) always terminates. The reason is that the logical predicates used to define the abstraction functions indeed partition the set of variable configurations of the thread variables. Provided that no threads are created in the loop, then the set of states reachable via the loop unrolling remains finite: imagine that we start with any subset of the (finitely many) states induced by the logical predicates, at every unrolling we either find new (abstract) transitions leading to new (abstract) states, or we reach a fixpoint of the unrolling operator. In the former case, we discover a larger set of reachable (abstract) states, which can be unrolled once more. Since the set of reachable (abstract) states is bounded by the set of all the abstract states (the latter being finite, as we just said), then the loop unrolling operation in REACHWHILE must always terminate.

Algorithm 6 Operator for the *procedure call*.

```

function REACHCALL( $stmt, s, \alpha$ )
   $c := called(stmt)$ 
  if  $c \in \{now, sleep_D, sleepUntil\}$  then
     $t := s.inc()$  ▷ reach the next LOC, state variables do not change
     $S := \{t\}$ 
     $T := \{s \rightarrow t\}$ 
  else
    if we have access to the source code of  $c$  then
       $S', T' := REACHTHREAD(c, \alpha)$ 
    else
       $S', T' := KB.getInterpretation(c, \alpha)$ 
     $u := s.push(0)$  ▷ enter the callee method's body
     $S'', T'' := shiftLOC(S', T', u.pc)$ 
     $I := initialStates(S'')$ 
     $F := finalStates(S'')$ 
     $S := S''$ 
     $T := T'' \cup \{s \rightarrow v : v \in I\} \cup \{w \rightarrow t : w \in F, t = w.pop().inc()\}$ 
  return  $S, T$ 

```

Assume operators $initialStates(S)$ (resp. $finalStates(S)$) returning the subset of locations in S that have no incoming edge (resp. no outgoing edge). Operator REACHCALL (see Algorithm 6) handles the case of a statement representing the invocation of either a timed function (introduced in Section 5) or a regular Java method. In the first case, we assume the untimed behavior is a dummy transition towards a new state where only the pc variable changes (increasing by one) while the other variables are untouched. In the case the callee is a regular Java method, then we assume to have an *interpretation* of it in a global dictionary that constitute a shared *knowledge base* (KB). We assume the interpretation of a method should be a timed automaton template describing the behavior of the method itself. Two cases are possible: if the source code of the callee is available, we invoke the REACHTHREAD(c, α) on it (see Algorithm 8) to build the timed automaton template from the code of the invoked method. This obviously creates a mutual recursion between REACHTHREAD and REACHCALL, and, in order to be well-founded, we must assume that every function/method call chain in the program code is non-recursive. Otherwise, if the source code of the callee is not available, then the user is responsible for providing the interpretation in the form of a timed automaton template whose nodes form a *bipartite* graph: input nodes have no incoming edges and they are connected to output nodes that have no outgoing edges. The edges and locations may specify additional time constraints.

In both cases, the locations of the looked up timed automaton template adjust the value of their pc component to perform a method inlining. They modify the template inserting the method body in a inner block right after the callee's value for pc . This is handled by the procedure *shiftLoc*.

Finally, the current location s is connected with an edge to every initial location of the method interpretation, and every final location w of such interpretation is connected to a location t where all variables keep the same value, with the exception of pc that is updated to the LOC immediately after the method invocation. This definition simulates the action of copy-and-pasting the callee method/function body in place of the method/function invocation in the callee. This heavily relies on the assumption that the verified code is non-recursive. We emphasize that when modeling a method invocation, we assume the correct type of the callee instance can be determined. While this is in contrast with the *Java virtual method invocation* principle, later we explain how additional user inputs and heuristic functions can help the methodology to solve such ambiguities related to dynamic typing rules.

In REACHTHREAD (see Algorithm 8), we give a procedure for building an untimed abstraction of a Java thread, starting from its code and an abstraction. The first step is to determine the initial abstract state, which is obtained by filtering all the abstract configurations of the attributes composing the abstraction α and keeping those configurations that fix the local and global variables of the thread to the expected initial value for their data type. Note that the thread parameters are allowed to assume any value in the initial state. Here, we consider the set of thread parameters to be composed of the attributes of the class implementing the thread itself, or the parameters passed to the run method that begins the execution of the thread itself.

In REACHTHREAD, we use transitions $s \xrightarrow{!!w_a} t$ to denote a special *broadcast send* transition, as meant by networks of timed automata (see Sec. 3.1). The label expresses the fact that jumping from (the abstract) state s to t , the (global) variable w has been updated to the new value a . A broadcast transition is well suited for modeling this kind of *visible* update, because in this way, every timed automaton in the network is forced to react with a complement transition $s' \xrightarrow{??w_a} t'$ added by ADDREACTEDGES (Algorithm 1), jumping from

(abstract) state s' to t' , such that $t' = s'[w \leftarrow a]$. In particular, note that $t'.pc = s'.pc$, i.e., the changed state does not reflect the execution of any statement (with consequent change in the pc value), but it only reflects a change in the global environment, while remaining at the same LOC.

Algorithm 7 Algorithm for finding the initial states.

```

function FINDINITIALSTATES( $\alpha$ , Parameters)
   $S_0 := \emptyset$ 
  for  $s$  in  $SS(\alpha, 0)$  do
     $is\_initial = true$ 
    for  $a$  in  $attributes(\alpha)$  do
      if  $a \notin Parameters$  and  $value(a)$  is not initial then
         $is\_initial = false$ 
        break
      if  $is\_initial$  then
         $S_0 := S_0 \cup \{s\}$ 
  return  $S_0$ 

```

Algorithm 8 Operator for handling *threads*.

```

function REACHTHREAD( $P$ ,  $\alpha$ )
   $S_0 := FINDINITIALSTATES(\alpha, Parameters(P))$ 
   $S := S_0, T := \emptyset, T' := \emptyset$ 
  for  $s$  in  $S_0$  do
     $S'', T'' := REACHSEQ(Body(P), s, \alpha)$ 
     $T' := T' \cup T''$ 
  for  $s \rightarrow t \in T'$  do
     $S = S \cup \{s, t\}$ 
    if  $t.w = a$  and  $s.w = a$  and  $w$  is a global variable then
       $T := T \cup \{s \xrightarrow{!!w_a} t\}$ 
    else
       $T := T \cup \{s \rightarrow t\}$ 
  return  $S, T$ 

```

Let us emphasize that almost all of the REACH-* rules make use of the SMT solver, through the ISSAT oracle. If we imagine to replace the invocation of ISSAT with an invocation of a dummy solver, always returning *true* to every input problem, the same rules would produce a set of *control-flow automata* abstracting the code under analysis. In control-flow automata, locations correspond to LOCs in the code and do not distinguish when the same LOC is hit twice in the program with very different configurations of the thread variables. This makes virtually impossible to model check interesting properties of real-world software, because:

- either the specification is given in terms of the thread variables configurations, or
- the control-flow automata have too many *spurious counter-examples*, i.e., two consecutive transitions in the abstract model falsify the given specification, but would never

be possible in the actual program, due to some conditional evaluation of the thread variables that are lost in the control-flow automata.

An example of this limitation will be shown in Section 8.

Lemma 4 *The procedure REACHTHREAD (Algorithm 8) always terminates.*

Proof We begin by observing that the REACH operators are recursively defined. Any sequence of recursive calls to REACH-* functions, though, reduces the size of the statement to be processed, with the exception of REACHWHILE and REACHCALL. If neither REACHWHILE nor REACHCALL is invoked along the sequence, then the sequence is obviously finite.

In case a REACHWHILE occurs in the sequence, we observe that the set of transitions produced at each step by REACHWHILE is monotonically increasing (because at every invocation, we preserve all the previously discovered transitions) and bounded from above (because the next computed set is always included in the set $SS(W) \times SS(W)$, which is finite due to the employed abstraction).

In case REACHTHREAD invokes (indirectly) REACHCALL, then the latter invokes REACHTHREAD again. Since we assumed that the verified code is not mutually recursive, every sequence of method calls in the verified code is finite. This implies that every mutually recursive sequence of REACHTHREAD-REACHCALL invocations is well defined. \square

6.2 Modeling complex data-structures

SMT solvers come equipped with several theories based on common data-types (e.g., integer numbers, real numbers, and bit vectors). Java programs, on the other side, almost always use data-structures more complex than SMT data-types, for which a theory has not been developed or is undecidable. The user of our approach and tool, then, needs a way to reduce an arbitrary Java data type onto an SMT one. To describe (an abstraction of) arbitrary Java data types, we exploit algebraic data types. Intuitively, a Java class definition is mapped onto an SMT record, collecting the *attributes* of the class itself, and a set of SMT functions, each modeling one of the Java methods. Each of such SMT functions takes as first argument an instance of SMT record denoting an instance of the Java class we are abstracting. For instance, the Java class `java.lang.String` can be abstracted with the following SMT record type:

```
(declare-datatypes () ((AbsString (init-AbsString
                                   (strval Int) (size Int))))))
```

i.e., a record called `AbsString` with a constructor named `init-AbsString`, and two fields `value` and `size`, both of types `Int`. While the meaning of field `size` is self-explanatory, it should be noticed that every string literal is associated with a numeric value by means of a (reverse-lookup) dictionary, i.e., every time a string literal, say "mickey", appears in the code (at compile time) a fresh integer value (say 1) is generated and associated with that string. Next, every occurrence of "mickey" in the code is replaced by a record `(init-AbsString 1 6)`, i.e., a record with value 1 and size 6.

A method such as `java.lang.String.equals` can then be mapped onto the following SMT predicate:

```
(= __return__ (= (strval __self__) (strval par.0)))
```

where `__return__` is an auxiliary variable for storing the (boolean) result of comparing the value of `__self__` and `par_0`, the former abstracting the current instance while the latter is linked to the (abstraction of) another string used for the comparison. This way of abstracting Strings is enough for checking equality of strings (it is enough to check that their values are the same) or to compare the lengths of two strings (by comparing their sizes). On the other side, it would not be a precise abstraction for different operations on strings, e.g., checking whether a string is contained within another.

Example 1 Suppose there is a Java method using the string literals "mickey" and "scrooge". Suppose the method contains the following conditional instruction: `if (a.equals("mickey")) { ... do something ... }` As a first step, such code is rewritten in the semantically equivalent one:

```
bool equals_1000 = a.equals("mickey");
if (equals_1000) { ... do something ... }
```

Next we have to check whether the guard can be satisfied. This is done through the following SMT problem:

```
(declare-datatypes () ((AbsString (init-AbsString (strval Int)
  (size Int))))))
(declare-const null AbsString)
(assert (= (size null) 0))
(assert (= (strval null) 0))
(declare-const a AbsString)
(assert (>= (size a) 0))
(assert (implies (= (strval a) 1) (= (size a) 6)))
(assert (implies (= (strval a) 2) (= (size a) 7)))

(declare-const MICKEY AbsString)

(assert (= MICKEY (init-AbsString 1 6)))

; begin encoding of current state
(assert (= a (init-AbsString 1 6)))
; end encoding of current state

; begin encoding guard: a.equals("mickey")
(declare-const equals_1000 Bool)
(assert (= equals_1000 (= (strval a) (strval MICKEY))))
(assert equals_1000)
; end encoding guard
(check-sat)
```

In the SMT problem, we encode the `AbsString` data-type, together with some constant (e.g., the interpretation of `null` and of `MICKEY`). Through some assertion, the tool restricts the set of coherent structures of type `AbsString`, i.e., those having non-negative value, and imposing that strings with value "mickey" must have size 6, while occurrences of "scrooge" must have size 7.

6.3 Abstracting time-dependent steps

Let us now introduce the notion of programs with timed behaviors. To do so, we assume that in addition to the underlying set of variables, a finite set of clock variables C exists. We also assume a family Γ of terms describing conditions on clock variables: $\Gamma ::= C \sim \mathbb{N} \mid C \sim C \mid \Gamma \wedge \Gamma$, where $\sim \in \{\leq, <, =, >, \geq\}$. Terms of Γ are also known as *clock conditions*.

Let us call *concrete timed program* (resp. *abstract timed program*) a state transition system $P = (S, S_0, T, C, I, G, R)$ such that (S, S_0, T) is a concrete (resp. abstract) program, $I : S \rightarrow \Gamma$ maps each discrete state to a clock condition also referred to as *time invariant*, $G : T \rightarrow \Gamma$ maps each discrete transition to one (possibly a tautology) enabling clock condition, and $R : T \rightarrow 2^C$ maps each discrete transition to zero or more clock variables to reset when taking the transition.

We call state sequence any finite or infinite sequence $(s_0, \gamma_0)(s_1, \gamma_1) \dots$ where $s_i \in S$ is called the discrete state and $\gamma_i : C \rightarrow \mathbb{T}$ is a clock valuation. Given a natural $\delta \in \mathbb{N}$ and a clock valuation $\gamma : C \rightarrow \mathbb{T}$, we will write $\gamma + \delta$ to denote the clock valuation where all clocks are advanced by the same amount δ . Given a set of clocks $X \subseteq C$, we will write $\gamma[X \rightarrow 0]$ to denote a new clock valuation $\gamma' : C \rightarrow \mathbb{T}$ such that $\gamma'(c) = 0$ if $c \in X$ and $\gamma'(c) = \gamma(c)$ if $c \in C \setminus X$.

A *timestamp* sequence is a sequence $t_0 t_1 \dots$ such that $t_{i+1} \geq t_i$, for all $i \in \mathbb{N}$, and $t_0 = 0$. We call *timed trace* any (possibly infinite) sequence $\rho = ((s_0, \gamma_0), t_0)((s_1, \gamma_1), t_1) \dots$ where $(s_0, \gamma_0)(s_1, \gamma_1) \dots$ is a state sequence, and $t_0 t_1 \dots$ is a timestamp sequence.

Assume a timed program $P = (S, S_0, T, C, I, G, R)$ and a timed trace $\rho = ((s_0, \gamma_0), t_0)((s_1, \gamma_1), t_1) \dots$. Then, the i th step in the trace $((s_i, \gamma_i), t_i)((s_{i+1}, \gamma_{i+1}), t_{i+1})$ is valid in P if one of the following holds:

- (*discrete step*) $\delta = 0 \wedge \tau \in T \wedge \gamma_i \models G(\tau) \wedge \gamma_{i+1} = \gamma_i[R(\tau)] \wedge \gamma_{i+1} \models I(s_{i+1})$
- (*timed step*) $\delta > 0 \wedge s_{i+1} = s_i \wedge \gamma_{i+1} = \gamma_i + \delta \wedge \gamma_{i+1} \models I(s_{i+1})$

where $\delta = t_{i+1} - t_i$ and $\tau = (s_i, s_{i+1})$. The trace ρ is a valid trace in P if every step in ρ is valid in P .

Notice that each (discrete or delay) transition requires that clock evaluation γ_{i+1} satisfies the clock condition $I(s_{i+1})$. This explains why $I(s_{i+1})$ is also called *time invariant* of state s_{i+1} .

Since Java does not provide a native type for clock variables, most Java programs keep track of the passage of time by means of integer timestamps, that are, from time to time, compared against other timestamps or the hardware clock. We assume, for each thread, a finite set of clocks $C = \{alive, sleep\} \uplus C_{deadlines}$, where $C_{deadlines}$ contains as many clock variables as the invocations of the $deadline_T$ operators used in the code. Intuitively, *alive* tracks the thread execution time and is always increasing. The single $sleep_D$ clock variable is sufficient to track the actions of entering, staying, and exiting the thread sleeping state, since each thread cannot have nested invocations of the $sleep_D$ function. Finally, since a thread can only define a finite number of $deadline_T$ operators, a finite number of clocks in $C_{deadlines}$ suffices. The restriction we imposed on the type of verified Java programs ensures that the number of nested blocks guarded by a $deadline_T$ is known statically. However, the actual values passed as arguments to $sleep_D$, $sleepUntil_T$, and $deadline_T$ cannot be determined statically, in general. In the following, we further restrict our setting assuming that arguments of $sleep_D$, $sleepUntil_T$, and $deadline_T$ are bounded by known intervals.¹⁰

¹⁰Such bounds could be given by the user or inferred by heuristic functions analyzing the code.

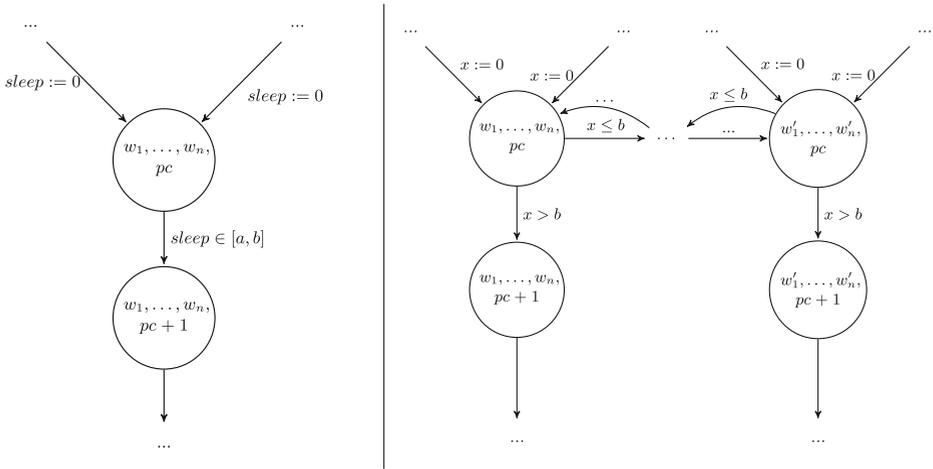


Fig. 6 A representation of modeling code that puts a thread to sleep (left) or checks for a deadline (right)

We argue that restrictions on such assumption are reasonable when modeling and verifying time-aware software systems. Parameters that affect the actual execution time of the code are critical for the correct timing of the code itself. Thus, they are usually specified as configurable parameters or determined at compilation time. In both cases, they can assume values within known intervals.

Assume a one-to-one mapping $clock : PC \rightarrow (C \cup \{\epsilon\})$ returning either the clock variable $sleep$ if $instr(pc)$ is a $sleep_D$ call, or a clock in $C_{deadlines}$ if $instr(pc)$ contains the expression $holds_T(deadline_T(v.1), v.2)$, for some Java variables v_1 and v_2 ,¹¹ otherwise it returns a distinguished symbol ϵ , denoting “no clock variable.” Let us write $in_{sleep}(s)$ iff $instr(s.pc)$ is a $sleep_D$ invocation and $in_{deadline}(s)$ iff $instr(s.pc)$ contains a $deadline_T$ instruction. Let us assume a mapping $bound(pc) \subseteq \mathbb{N}$ for any $pc \in PC$, such that $bound(pc)$ evaluates to a non-empty convex interval if $clock(pc) \neq \epsilon$; otherwise, it returns an empty interval.

Intuitively, if the statement at LOC pc has the form $sleep_D(v.i)$, the interval $bound(pc)$ is expected to contain the actual value of variable v_i . On the contrary, if the statement at the LOC pc contains the expression $holds_T(deadline(v.1),v.2)$, then the interval $bound(pc)$ is expected to contain the actual value of every evaluation of the difference $v_1 - v_2$.

Figure 6 contains an intuitive explanation of how pieces of programs are translated onto (pieces of) timed automata. On the *left*, it shows how to model a call to the `sleep` function at some code position $pc \in PC$ with its approximated duration interval $[a, b] = bound(pc)$. This ensures that the control stays in the current state for at least a time units and will leave in at most b time units.

In the figure, w_1, \dots, w_n, pc denotes the abstract discrete state where the $sleep_D$ call happens, having the following state invariant: $I(w_1, \dots, w_n, pc) = (sleep_D \leq b)$.

¹¹Following Section 5, the only meaningful use of such pattern is as guard of a conditional or loop statement.

On the *right*, the figure shows a set of states and transitions simulating the behavior of a statement of the form: $\text{while } (\text{holds}_T(\text{deadline}_T(v.1), v.2)) \{ \dots \}$, at some position pc and such that $\text{bound}(pc) = [a, b]$. There, $x = \text{clock}(pc)$ represents the (only) clock variable associated with the deadline statement at position pc in the code. The REACHWHILE rule can unroll the while loop onto a sub-graph of reachable states and transitions, several of which are at LOC pc , each re-evaluating the deadline guard. Each such location must decide whether to enter the body of the while statement or skip it, jumping to a location where the discrete variables are left untouched, but the LOC changes to the value returned by $\text{inc}_{TD}(pc)$.

Algorithm 9

```

function BUILDNTA ( $P_1, \dots, P_n, \alpha$ )
  for  $i$  in  $1..n$  do
    //  $P_i$  is the  $i$ -th thread of the Java program
     $(\hat{S}, \hat{T}) := \text{REACHTHREAD}(P_i, \alpha)$ 
     $\hat{S}_0 := \text{initialStates}(\hat{S})$ 
     $\hat{C} := \text{cod}(\text{clock}) \setminus \{\epsilon\}$ 
     $\hat{I} := \{\text{INVARIANT}(s) : \hat{s} \in \hat{S}\}$ 
     $\hat{G} := \{\text{CLOCKGUARD}(s, t) : (s, t) \in \hat{T}\}$ 
     $\hat{R} := \{\text{CLOCKRESET}(s, t) : (s, t) \in \hat{T}\}$ 
     $\hat{P}_i := (\hat{S}, \hat{S}_0, \hat{T}, \hat{C}, \hat{I}, \hat{G}, \hat{R})$ 
  return  $(\hat{P}_1, \dots, \hat{P}_n)$ 

```

Assume three operators: $\text{CLOCKGUARD} : T \rightarrow \Gamma$ returns a clock constraint to be checked before taking a transition, $\text{CLOCKRESET} : T \rightarrow 2^C$ returns the set of clock variables to be reset at each transition, and $\text{INVARIANT} : S \rightarrow \Gamma$ returns a clock expression that must be satisfied at every moment in time by the state. Below, we give their definitions, for any possible states $s, t \in \hat{S}$.

$$\begin{aligned}
 \text{INVARIANT}(s) &= \begin{cases} \text{sleep} \leq b & \text{if } \text{in_sleep}(s) \text{ and } \text{bound}(s.pc) = [a, b] \\ \text{true} & \text{otherwise} \end{cases} \\
 \text{CLOCKGUARD}((s, t)) &= \begin{cases} \begin{cases} \text{sleep} \in [a, b] & \text{if } \text{in_sleep}(s.pc) \text{ and } \text{bound}(s.pc) = [a, b] \\ x \leq b & \text{if } \text{instr}(s.pc) = \text{"if (holds(dl))"} \wedge \\ & t = s.\text{push}(\text{THEN}).\text{push}(0) \wedge \\ & x = \text{clock}(s.pc) \wedge \\ & \text{bound}(s.pc) = [a, b] \end{cases} \\ x \geq a & \text{if } \text{instr}(s.pc) = \text{"if (holds(dl))"} \wedge \\ & t = s.\text{push}(\text{ELSE}).\text{push}(0) \wedge \\ & x = \text{clock}(s.pc) \wedge \\ & \text{bound}(s.pc) = [a, b] \\ \text{true} & \text{otherwise} \end{cases} \\
 \text{CLOCKRESET}((s, t)) &= \begin{cases} \{\text{sleep}\} & \text{if } \text{in_sleep}(t) \\ \{x\} & \text{if } \text{in_deadline}(s) \text{ and } \text{clock}(s.pc) = x \\ \emptyset & \text{otherwise} \end{cases}
 \end{aligned}$$

In Alg. 9, we show the procedure BUILDNTA extending the finite-state representation of the threads to a network of timed automata. From Lemma 4, it immediately follows that Alg. 9 terminates as well.

7 Soundness

As shown in Section 4, the Java language has many dynamic features such that common static analysis problems fall in the undecidable fragment. Any hope for a complete static and automatic analysis targeting the totality of Java programs is thus doomed to failure.

At this stage, we wish to establish the soundness of our static analysis for Java programs that fall in a static subset of the language that we refer to as *kernel-Java* in the following. We recognize the following characteristics of the Java language that easily lead to intractability, when doing static analysis:

- threads can be created and destroyed at run-time: several works on parameterized verification showed that already the reachability problem of a system with an unknown number of copies of finite-state threads is undecidable (the interested reader can find an overview on the topic in Aminof et al. (2018) for untimed systems and Spalazzi and Spegni (2020) for timed systems);
- recursive or mutually recursive method calls can generate an unbounded number of records in the activation stack;
- when calling a method, the exact reference of the method declaration to be invoked is determined at run-time, due to the well-known *dynamic dispatching*: we could generate an (very large) over-approximation of the program where each *callee* method non-deterministically picks any *called* method with that signature, at every invocation, but this would cause an explosion of the state-space to be explored;
- the Java language has a rich type system, resulting in an undecidable type-checking problem (Grigore 2017).

As a consequence, we define *kernel-Java* to be the subset of Java where, at compilation time:

- the set of running threads is fixed;
- recursive or mutually recursive method calls are not allowed;
- for each method call, the invoked method body is determined;
- for every object, we can determine its exact type.

The nature of *kernel-Java* is notably that of a static language (similar to previous efforts, e.g., Java-light or Bali (Nipkow and Von Oheimb 1998)). Furthermore, any *kernel-Java* thread, due to the severe restrictions that we impose on its structure, can be rewritten onto an equivalent Java thread where method invocations have been replaced by the method body itself with minor adjustments due to variable renaming in order to simulate the passage of arguments when invoking the method itself, and receiving the returned value at the end of the invocation. We also assume that any complex expression appearing as guard of an if-then-else or while statement, as well as method arguments, is unrolled in the natural way and their result assigned to *auxiliary fresh variables* that are then used as guards for the conditional or loop statement, or passed as arguments to the method, respectively.

In the following, we prove the soundness of our procedure for extracting networks of timed automata assuming that the original threads have already been translated onto a set of *kernel-Java* threads composed of the following Java control structures:

- variable declarations and assignments;
- sequence of statements;
- conditional statements (in the form of if-then-else) guarded by a variable;

- loops (in the form of while statements) guarded by a variable;
- invocations of methods whose source code is not in the repository.

Since many interesting programs with non-finite and timed behavior still fall in this class, here we show to what extent timed specifications of the original program are preserved by the abstraction.

Given a program P , we call *execution* $\chi = s_0 r_1 r_2 \dots$ an initial kernel-Java configuration s_0 followed by a (possibly infinite) sequence of Java rules $r_1 r_2 \dots$ applied one after the other. We also write χ_0 denoting the initial configuration s_0 and χ_i , for $i \geq 1$, to denote the i th Java rule applied along χ .

A *timed trace* of program P is, instead, a (possibly infinite) sequence $\rho = (s_0, t_0)(s_1, t_1) \dots$ of program configurations and time values, induced by some execution χ . More precisely, given an execution χ , the initial configuration is $s_0 = \chi_0$ and the initial time value $t_0 = 0$, while the i th configuration, for $i > 0$, is $s_{i+1} = \chi_i(s_i)$ i.e., the configuration resulting from applying the Java rule χ_i to configuration s_i ; if χ_i is a tick rule, then $t_{i+1} = t_i + 1$, else $t_{i+1} = t_i$.

We call *abstracted method* a method for which the user provided an interpretation in the form of a timed automaton template; otherwise, it is *forgotten*. Assume a program P , an abstraction α , and a set of forgotten methods F . We say the pair (α, F) is a *sleep-precise* abstraction of P if no forgotten method contains any invocation of the $sleep_D$ function. Similarly, we say the pair (α, F) is a *deadline-precise* abstraction of P if no forgotten method contains any $holds_T$ expression on a deadline. Intuitively, we call deadline-precise and sleep-precise those abstractions that do not loose meaningful information about their deadlines and delays. Note that, when an interpretation of the method is present, either the REACHTHREAD generated it; thus, it is precise *by construction*, or the user provided it, in which case the tool *assumes* it is precise. Let us write $P = (P_1, \dots, P_n)$ to denote the fact that program P is the composition of threads P_1, \dots, P_n . A network of timed automata $nta = BuildNTA(P_1, \dots, P_n, \alpha)$ is a *sleep-precise* (resp. *deadline-precise*) abstraction of P if the pair (α, F) is a *sleep-precise* (resp. *deadline-precise*) abstraction of P .

Let us now introduce a notion of simulation between a program and a network of timed automata. It will be used later in order to show that our methodology, given a program, does not produce an *arbitrary* network of timed automata, but one that *simulates* the program behavior.

Definition 1 (Simulation) Given a kernel-Java program P and a network of timed automata nta , we say that nta *simulates* P (written $P \preceq nta$) if there is an abstraction α such that:

- for every configuration s in P , $\alpha(s)$ is a state in nta ;
- for every transition $s \xrightarrow{Tick} s'$, where s and s' are configurations, there exists a timed step $\alpha(s) \xrightarrow{\delta} \alpha(s')$ in nta where $\delta = 1$;
- for every other transition $s \xrightarrow{r} s'$, where s and s' are configurations and r a Java rule different from $Tick$, there exists a discrete or broadcast step $\alpha(s) \xrightarrow{i_1, \dots, i_n} \alpha(s')$ in nta .

Given two timed traces $\rho = (s_0, t_0)(s_1, t_1) \dots$ and $\rho' = (s'_0, t'_0)(s'_1, t'_1) \dots$, we say that ρ' *corresponds* to ρ modulo some abstraction α , written $\rho' = \alpha(\rho)$, if it holds that $t'_i = t_i$ and $s'_i = \alpha(s_i)$, for all $i \geq 0$.

Lemma 5 *Given a kernel-Java program P and a network of timed automata nta simulating P (i.e., $P \preceq nta$), then the following holds:*

$$\forall \varphi \in \text{MTL}. nta \models \varphi \Rightarrow P \models \varphi$$

Proof This theorem adapts the well-known simulation theorems for untimed and timed systems to kernel-Java programs and timed automata.

First of all, let us clarify that $P \models \varphi$ (resp. $nta \models \varphi$) means that for any timed trace ρ in P (resp. ρ' in nta), then the timed trace satisfies the formula, i.e., $\rho \models \varphi$ (resp. $\rho' \models \varphi$).

By definition of simulation, given any timed trace ρ of program P , the corresponding trace $\rho' = \alpha(\rho)$ is a timed trace as well, since every step in P admits a corresponding step in nta . Since the satisfaction relation of MTL formulae is defined upon timed traces, one can check (by structural induction on the MTL formula) that $\rho' \models \varphi$ implies $\rho \models \varphi$.

Since this holds for any timed trace, the statement follows. \square

The following lemma is a straightforward extension of a very similar result on simulation-equivalent timed systems and ATCTL formulae (Konnov et al. 2017).

Lemma 6 *Given a kernel-Java program P and a network of timed automata nta simulating P (i.e., $P \preceq nta$), then the following holds:*

$$\forall \varphi \in \text{ATCTL}. nta \models \varphi \Rightarrow P \models \varphi$$

Theorem 1 (Simulation) *Assume a kernel-Java program $P = (P_1, \dots, P_n)$, an abstraction α , and a network of timed automata $nta = \text{BuildNTA}(P_1, \dots, P_n, \alpha)$ such that nta is a sleep-precise and deadline-precise abstraction of P . Then $P \preceq nta$.*

Proof First of all, we focus on relevant parts of configurations of program P : in it every thread has the following cells $\langle \text{Map} \rangle_{\text{env}}$, $\langle \text{Map} \rangle_{\text{store}}$, $\langle \text{Nat} \rangle_{\text{time}}$, $\langle \text{Nat} \rangle_{\text{sleep}}$, $\langle \text{List} \rangle_{\text{deadlines}}$. Cells `env` and `store` are inherited from the KJ semantics; the former associates variable names with store locations, while the latter associates store locations to actual values. The other cells have been described in detail in Section 5. We can abstract away this complexity saying that each thread in the program has a finite set of variables V tracking the values of variables and time values. Next, the thread configuration s can be represented with tuples $(a_1, \dots, a_n, k_1, k_2, d_1, \dots, d_m)$ if variable v_i in configuration s is mapped (through the store) to value a_i , for $i \in [1, n]$, the value of cell `time` is k_1 , the value of cell `sleep` is k_2 , and the j th element on the list `deadlines` has value d_j , for $j \in [1, m]$. Since we restricted our analysis to the kernel-Java subset of the Java language, the set of all program configurations is statically definable and is a subset of $SS(V)$, where $V = \{v_1, \dots, v_n\}$ is the set of thread variables. Note that the set of thread variables includes both local and global variables.

Since we assumed ta to be a sleep-precise and deadline-precise abstraction of P , there must exist a set of forgotten methods F such that the pair (α, F) is a sleep-precise and deadline-precise abstraction of P . By definition of α , its codomain is an abstract state-space $SS(W)$ containing the abstract states of ta , satisfying the first requirement for a simulation.

Now, let us take any step $((a_1, \dots, a_n, k_1, k_2, d_1, \dots, d_m), t) \xrightarrow{r} ((a'_1, \dots, a'_n, k'_1, k'_2, d'_1, \dots, d'_m), t')$ pushing the program configuration $(a_1, \dots, a_n, k_1, k_2, d_1, \dots, d_m)$ at time t to a new configuration $(a'_1, \dots, a'_n, k'_1, k'_2, d'_1, \dots, d'_m)$ at time t' , after applying the timed semantic rule r .

In the case rule r is the TICK rule, then $t' = t + 1$. By definition, $a_i = a'_i$, for $i \in [1, n]$, and $k'_1 = k_1 + 1$, $k'_2 = k_2 + 1$, and $d'_i = d_i + 1$, for $i \in [1, m]$, i.e., all time values advanced by one time unit. On the template automaton side, a delay transition with $\delta = 1$ causes the step: $\alpha(a_1, \dots, a_n, k_1, k_2, d_1, \dots, d_m) \alpha(a_1, \dots, a_n, k_1 + 1, k_2 + 1, d_1 + 1, \dots, d_m + 1)$, satisfying the second requiring for a simulation.

In the case rule r is not the TICK rule, then $t' = t$, and r is the interpretation of some statement ι of kernel-Java. Next, reasoning by cases, one shows that the concrete step in the kernel-Java program P is mimicked by an enabled abstract transition in the network of timed automata nta .

Consider the case when r is the rule LOCALVARDEC. Such rule is applied if there exists a thread whose code is currently declaring a local variable in kernel-Java. The rule consumes the current statement (a variable declaration, indeed), then it updates the env cell by linking the variable name v to a fresh address L in the store cell, it sets address L in the store cell to point to the initial value for its type ($\text{unruled}(\text{type}(T))$), and finally it increases a counter in the cell `nextLoc`, responsible for generating fresh addresses at each lookup. Since in kernel-Java we assumed that variable initialization in variable declarations is moved into a subsequent assignments, we do not have to cover that case. Since we restricted to a static subset of the language, we assume a fixed environment and store, where all the names and addresses are initialized to the default abstract value for the given type. Thus, it is enough to stutter in the abstract state $(\alpha(a_1, \dots, a_n), k_1, k_2, d_1, \dots, d_m)$ in order to simulate this kernel-Java transition.

Consider the case when r is the rule ASSIGN. Such rule is applicable when a kernel-Java thread assigns an r -value w to the location in the store corresponding to variable v_i . Since we are working with kernel-Java, the r -value was obtained by looking up a variable, an object field, or is a literal written in the right-hand side of the assignment statement itself. Note that in kernel-Java, the r -value must be of some basic type T_1 , while the location has some type T_2 . In this case, rule r first checks that type T_1 is a sub-type of T_2 , and next it assigns the r -value to the specified location. In our approach, REACHASSIGN tests whether some abstract state t exists such that $\text{ISAT}(\text{predicate}(s) \wedge \llbracket \text{stmt} \rrbracket_{\text{SMT}} \wedge \text{indexed}(\text{predicate}(t)))$, where stmt is the assignment statement interpreted by rule ASSIGN. Since, by construction, REACHASSIGN tests such property for every possible (abstract) state t in the state-space, and since by assumption $(a_1, \dots, a_i, \dots, a_n, k_1, k_2, d_1, \dots, d_m) \xrightarrow{\text{Assign}} (a_1, \dots, a'_i, \dots, a_n, k_1, k_2, d_1, \dots, d_m)$, where $a'_i = w$, and $s = (\alpha(a_1, \dots, a_i, \dots, a_n), k_1, k_2, d_1, \dots, d_m)$, then at least one such t must exist, viz. $t = (\alpha(a_1, \dots, a'_i, \dots, a_n), k_1, k_2, d_1, \dots, d_m)$ and transition $s \xrightarrow{(v_i = w)!!} t$ is enabled in the thread. Note that, following the definition of timed automaton template given in Section 6.3, the symbol $(v_i = w)!!$ denotes that the current transition is a sending-broadcast transition. By construction, if v_i is a global variable, all timed automaton templates have, for any location, an internal transition labeled with $(v_i = w)??$ and *react* to the change of a global variable, updating their location accordingly. This ensures that any side effect of the Java ASSIGN rule is simulated by the corresponding timed automaton transition.

Consider the case when r is the rule IFTRUE. This rule has been applied because one kernel-Java thread executed the conditional statement whose guard was a variable that evaluated to True (remember that in kernel-Java we only consider if-then-else statements whose guards are variables). In this case, the next thread instruction would be the body of the then branch (that will be either a block or a single statement). Thus,

it must be that $(a_1, \dots, a_n, k_1, k_2, d_1, \dots, d_m) \xrightarrow{\text{IfTrue}} (a'_1, \dots, a'_n, k_1, k_2, d_1, \dots, d_m)$ and that $\text{ISSAT}(v_1 = a_1 \wedge \dots \wedge v_n = a_n \wedge \text{guard}(\text{stmt}))$.

Call $s = (\alpha(a_1, \dots, a_n), k_1, k_2, d_1, \dots, d_m)$. Our assumptions imply that $\text{ISSAT}(s \wedge \text{guard}(\text{stmt}))$ holds; thus, the procedure REACHITE adds the transition $s \rightarrow t$, for $t = (\alpha(a'_1, \dots, a'_n), k_1, k_2, d_1, \dots, d_m)$ to nta . A symmetric reasoning is applicable in the case of the rule `IfFalse`.

For the other constructs of the Java language, by very similar arguments, we can show that kernel-Java rules are mimicked by abstract transitions computed by our methodology. \square

Theorem 1, Lemmas 5 and 6 yield the following results.

Corollary 1 *Assume a kernel-Java program P , a network of timed automata nta that is a sleep-precise and deadline-precise abstraction of P . Then $\text{nta} \models_{\text{NTA}} \varphi \Rightarrow P \models_{\text{kJ}} \varphi$, for any φ in $\text{MTL} \cup \text{ATCTL}$.*

8 Experimental validation

We have implemented in a prototype tool the interactive abstraction and verification methodology presented in Section 6. In the tool, the user specifies the set of threads he/she wants to abstract and a set of first-order predicates over program variables. He/she also specifies the temporal properties that should be checked and any additional temporal constraints that are known from the real-time assumptions of the environment where the threads are supposed to run. The overall task can be seen as the combination of several static analyses sub-tasks. A graphical overview of the interaction model implemented by the tool is given in Fig. 7.

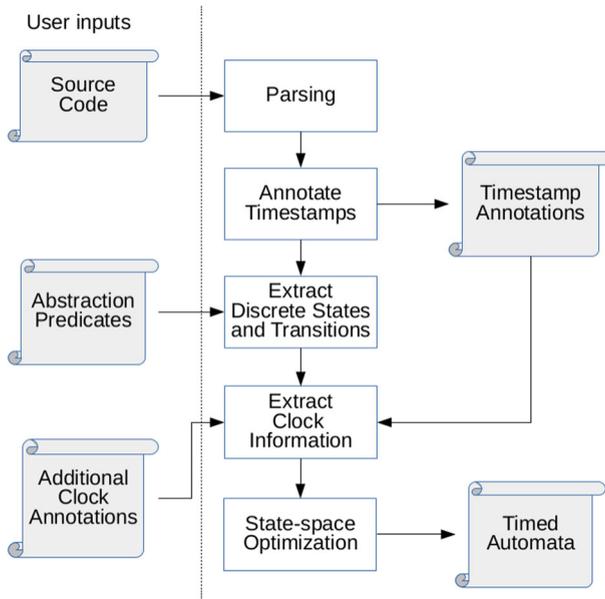


Fig. 7 Our methodology at a glance

The *parsing* step consists in extracting an *intermediate representation* of the entire Java project. We exploit the Eclipse JDT parser for Java 8 to produce a reduced abstract syntax tree (AST) from the code, and we store it into a no-sql database for saving time when the methodology is used interactively by the user.

The successive phase traverses the AST and along the way it *annotates timestamp variables*. Inspired by Liva et al. (2017), and using a list of common Java methods manipulating timestamps as well as Java types used to represent time values, we label as *timestamp variables* those variables in the program that are used as timestamps along the program (e.g., because they store the result of method `java.lang.System.currentTimeMillis`, or because they are passed as input to the `java.lang.Thread.sleep` method).

We expect that the list of Java methods and types used to identify timestamps in a program can be maintained in a centralized way, together with the implementation of the methodology itself. Furthermore, users can add custom types and methods and extend this list. Ideally, finer implementations of the methodology can allow communities of users to share their customizations, and the knowledge acquired along the analysis of Java software.

The next step focuses on *extracting discrete states and transitions* representing the program discrete behavior. To this aim, we require the user to provide a set of first-order predicates over a subset of the program variables. Through them, it is possible to abstract each concrete configuration of the program variables onto a single first-order predicate. If one or more variables have no associated predicate, we assume they can be assigned any value. The predicates specified by the user can look at a single thread variable (e.g., $x < 0$, $x \geq 0$), or they relate the concrete values of multiple thread variables at the same time (e.g., $x < y$, $x \geq y$). Successively, each instruction ι of the program is interpreted as a first-order predicate $\alpha(\iota)(s, t)$, relating the abstract state of variables *before* executing the given instruction (s), to the abstract state of the same variables *after* executing it (t). Notice that, in general, it is not possible to give a first-order interpretation of any arbitrary Java instruction.¹² For this reason, this step employs a set of rules that can be extended over time to detect relevant patterns appearing in Java programs. In case that none of the rules applies to the Java instruction under analysis, we map the instruction onto the tautology binary predicate $\alpha(\iota) = \top$, relating any source configuration to any target configuration. This ensures that every abstract transition $\alpha(\iota)$ is an *existential abstraction* (as seen in Section 6) of the concrete instruction ι , for any ι . To check that there can exist a transition $\alpha(\iota)$ from s to t , we use Z3, a state-of-the-art SMT solver (De Moura and Bjørner 2008).

The successive phase *extracts timing information* encoded in the program. This is achieved by identifying a suitable set of clock variables tracking the time relations between events as they are handled by the program. Since the final model will be a network of timed automata, this consists in inferring:

- the *clock variables* of each timed automaton,
- the *clock constraints* enabling the transitions of each timed automaton, and
- the discrete transitions *resetting* the clock variables.

This stage takes advantage of the timestamp annotations added in the previous stage, together with additional *clock annotations* added by the user.

We implement a final *state-space optimization* step similar to large-block encoding (Beyer et al. 2009). In it, sequences of transitions that do not branch and differ only for the value of the *program-counter* are collapsed into a single transition.

¹²Think for example to the invocation of a recursive method on some arbitrary input.

This simple optimization is already proven to be very helpful in reducing the size of the extracted timed automata.

The network of timed automata that results from applying our methodology can be used for several purposes, e.g.:

- for model checking safety and security policies against some logical properties provided by the user (e.g., using Uppaal (Larsen et al. 1997));
- for simulation purposes (e.g., using Uppaal); and
- as a documentation, giving a high-level view of the code (e.g., for software (re-) engineering purposes).

Let us emphasize that the methodology is designed to be interactive: if the user finds the network of timed automata that has been returned not to be precise enough for checking the desired security policy, he or she can change the list of abstraction functions and generate a more refined discrete component, or alternatively add more detailed clock information about the time handling of events by the program itself.

We studied the methodology using our prototype tool on three use cases: a Java implementation of the Fischer’s algorithm, presented in Fig. 5, and the code taken from two reported time bugs of two different open source projects, namely Apache Kafka and Alluxio.

Fischer’s algorithm. For this case study, we used the following predicates for abstracting the configuration of variables in the program:

- $x = 0, x = 1, x = 2, x > 2$
- $y = 0, y = 1, y > 1$
- $id = \text{‘foo’}, id = \text{‘fie’}$

We instructed the tool to model check the Fischer’s algorithm in a system with two threads, say p and q . We also specified, through the tool language, a time constraint that cannot be inferred from the source code, viz. that it takes less than DELTA time units to go from LOC 0.0 to LOC 0.1 (see Fig. 5), i.e., from testing $x \neq \text{null}$ to setting $x = \text{this.id}$. Such constraint is a known physical requirement for the Fischer’s algorithm to ensure mutual exclusion (Lampert 1987). Next we model checked the ATCTL specification given in Fig. 8.

Formulae `is_foo` and `is_fie` check that an arbitrary process, say p , can either assume the identifier “foo” or “fie.” Formula `p_q_diff` checks that the two threads, p and q , can assume different identifiers. Formula `good` checks that the shared variable y may assume value 1. Formulae `mutex` and `mutex2` are alternative encodings of the mutex property. Formula `nstarve` encodes the usual property of absence of starvation, where a generic thread p is checked to eventually reach LOC 4, the location in which the thread terminates.

Based on the given abstraction predicates, the tool generates a timed automaton template with 235 locations and 1154 edges. Two clock variables are extracted, viz. `C_PROG` and `C_CONSTRAINT`: the former is used to ensure that the sleep time is exactly DELTA time

$$\begin{array}{ll}
 \text{is_foo:} & \mathbb{E} F_{\geq 0} (p.id = \text{‘foo’}) \\
 \text{is_fie:} & \mathbb{E} F_{\geq 0} (p.id = \text{‘fie’}) \\
 \text{p_q_diff:} & \mathbb{E} F_{\geq 0} (p.id \neq q.id) \\
 \text{good} & \mathbb{E} F_{\geq 0} (y = 1) \\
 \text{mutex:} & \mathbb{E} G_{\geq 0} (y \leq 1) \\
 \text{mutex2:} & \mathbb{E} G_{\geq 0} (y > 1 \Rightarrow p.id = q.id) \\
 \text{nstarve:} & \mathbb{A} F_{\geq 0} (p.pc = 4)
 \end{array}$$

Fig. 8 ATCTL specification for Fischer’s algorithm

units, while the latter is used to bound the time between testing for variable x and resetting it in less than DELTA time units.

Our tool is able to find a counterexample for all the properties shown in Fig. 8, but `mutex2`. In particular, for the `mutex` formula, it is enough that both processes start with the same name (e.g., “foo”) in order to have two processes at the same time in the critical section. Instead, the more stringent formulation of the `mutex` property, viz. `mutex2` holds. It is a natural assumption that two threads do not share their identifier, but at the same time, this assumption cannot be inferred from the code, but must be provided by the user as part of the specification (see specification `mutex2`). In order to test the actual correctness of the extracted network of timed automata, we checked that known bugs in the Java code are correctly identified by our tool. To this aim, we performed two tests that we expect to falsify the specification:

- in the first test (V1), we kept the verification script and we changed the Java code to increment the shared counter y when $pc = 1$, but we commented out the line where the same variable is decremented (i.e., at location $pc = 2$ in Fig. 5);
- in the second test (V2), we kept the Java code but we relaxed the time assumptions under which the system should be verified, i.e. dropping the assumption that testing the value of variable x and setting it should require strictly less than DELTA time units.

The tool is able to discover two different counterexamples for the `mutex2` property in (V1) and (V2): in the former, the counterexample involves 17 Java instructions showing that one thread enters the critical section, but due to the fact that now y can only be increased, the state where $y > 1$ is reached, which in turn falsifies the specification. On the contrary, in (V2), a more critical bug is reported, due to the fact that the following interleaving is possible:

- p_1 tests that $x = \text{null}$ ($pc = 0.0$)
- p_2 tests that $x = \text{null}$ ($pc = 0.0$)
- p_1 sets $x = \text{"foo"}$ ($pc = 0.1$) and then starts sleeping ($pc = 0.2$)
- p_1 ends sleeping ($pc = 0.3$), checks that `this.id.equals(x)` ($pc = 1$), and increment variable y ($pc = 2$)
- p_2 sets $x = \text{"fie"}$ ($pc = 0.1$), then starts sleeping ($pc = 0.2$), it ends sleeping ($pc = 0.3$), checks that `this.id.equals(x)` ($pc = 1$), and increment variable y ($pc = 2$).

At the end of this path, the automaton reaches a state where $y > 1$ holds.

Our tool finds a correct counterexample for `nstarve`, where a process, say p , repeatedly obtains access to its critical section, while the other, q , cannot progress. This is a known limitation of the Fischer’s algorithm for mutual exclusion (Lamport 1987).

Let us observe that, as we anticipated in Section 6, in the case we abstract the code under analysis with a (set of) control-flow automata, the core specification of the Fischer’s algorithm, i.e., `mutex2`, would be falsified by a spurious counter-example, due to the concatenation of two transitions from subsequent LOCs even though the conditions on the thread variables would never allow such jumps in the real code. Indeed, we know that the Java implementation of the Fischer’s algorithm satisfies specification `mutex2`. This issue should not be confused, though, with the falsification of specifications `mutex` and `nstarve`, described previously: the former is falsified because of lack of information, i.e., the system cannot infer from the code that two threads will never share the same identifier; the latter is falsified because it is known that the Fischer’s algorithm can, in principle, cause

```

public void poll(long timeout) {
    // poll until the timeout expires
    long now = time.milliseconds();
    long deadline = now + timeout;

    while (now <= deadline) {
        if (coordinatorUnknown()) {
            ensureCoordinatorReady();
            now = time.milliseconds();
        }

        if (needRejoin()) {
            ensureActiveGroup();
            now = time.milliseconds();
        }

        pollHeartbeat(now);
        long remaining = Math.max(0, deadline - now);
        client.poll(Math.min(remaining, timeToNextHeartbeat(now)));
        now = time.milliseconds();
    }
}

```

Fig. 9 Example of real-time Java code containing a security bug

two threads to loop infinitely while trying to get access to their critical sections. In practice, this is an accepted behavior because “such starvation is unlikely to occur” (Lamport 1987).

Apache Kafka. A second verified piece of code is reported in Fig. 9. In this example, the method is the core of a Java thread of the Apache Kafka project, a popular distributed streaming platform allowing to implement a publish-subscribe service to streams of data.

The method `poll` implements a poll mechanism, where a server is checked periodically, and if it is not in a “ready” state, the `ensureCoordinatorReady` operation is invoked. This method contained a bug¹³ appearing when the parameter `timeout` assumes a negative value, or a big enough value, such that expression `now + timeout` evaluates to a value smaller than `now` (e.g. due to integer overflows). In this case, the presence of a bug can be detected by analyzing a single thread running that piece of code. By using our prototype tool, the user can specify that two abstract variables should be used, viz. `is_ready` and `coordinator_known`. Then, the user specifies the following first-order interpretation of method `ensureCoordinatorReady()`: `(assert (= is_ready_1 true))`, while method `coordinatorUnknown` is abstracted as follows: `(assert (= __return__ coordinator_known))`, where `__return__` is an auxiliary SMT variable used to store the result of the method invocation. All other methods are abstracted with a first-order tautology, meaning that they have no effect on the variables `is_ready` and `coordinator_known`. Finally, the user specifies that he/she wants to verify a system with only a single instance of the `poll` timed automaton template. The tool automatically recognizes two timestamps, viz. `deadline` and `now`. The number of states in the timed automaton template is 204 states, i.e., 4

¹³Bug: <https://issues.apache.org/jira/browse/KAFKA-4290>.

Table 1 Summary of experimental data

	SMT queries	Locations	Edges	Clocks	Specs	Time	Memory
Fischer	10,044	235	1154	2	7	2 min 54 s	100.1 MiB
Fischer buggy	8100	217	1064	2	7	2 min 07 s	96.1 MiB
Kafka	7140	204	612	1	1	8 s	45.2 MiB
Alluxio	22,342	684	4873	1	1	1 min 32 s	77.3 MiB

configurations of the two boolean variables `is_ready` and `coordinator_ready`, times the 17 values of the program-counter register, times the 3 possible abstract values of parameter `deadline`: `deadline < 0`, `deadline = 0`, and `deadline > 0`. The timed automaton contains one clock variable now used to track the difference between timestamps `deadline - now`, while the timestamp `deadline` yields a constant parameter with the same name that is added to the timed automaton template. The correctness requirement can be encoded with the following ATCTL formula: $\mathbb{A}F_{\geq 0}(is_ready = true)$. The counterexample found by Uppaal is the following: $(\sigma, pc = 0) \rightarrow (\sigma, pc = 1) \rightarrow (\sigma, pc = 2) \rightarrow (\sigma, pc = 3)$, where $\sigma := deadline < 0 \wedge is_ready = false \wedge coordinator_ready = true$. A simple code inspection allows to understand that such counterexample is not spurious, i.e., it is not added by the abstraction process, but it can happen with concrete executions of the method.

Alluxio. A third test bench experiment is conducted on the `acquire` method of class `alluxio.resource.DynamicResourcePool` of the Alluxio project. The method `acquire` accepts a timeout parameter that expresses the maximal amount of time that the caller is willing to wait for acquiring a resource. The method implements the acquisition with a `while (true) { ... }` loop that iterates until either the resource is acquired or it times out throwing an exception. A variable `endTimeMs` contains the expiration date that is used to verify whether the request times out. It is computed as the sum of current time and the timeout parameter. Since there is no check that the latter receives a negative value, it can happen that the `acquire` method never actually attempts to acquire the resource. Thus, the method wrongly returns the timeout exception without waiting for the resource to be available.¹⁴

In this case, the extracted timed automaton template counts 259 states, 381 transitions, and 1 clock variable. The checked specification is $\mathbb{A}F_{\geq 0}(is_healthy = true)$ and it is falsified by a counterexample assigning a negative value to the input parameter.

Methodology evaluation. In Table 1, we show some data collected from the experimental validation. Even though the limited number of case studies does not allow us to make a quantitative evaluation of the methodology, they already provide a qualitative feedback about how practical it is, when applied to real-world software projects. First of all, one of the strengths of the methodology, i.e., the fact that the user can specify the abstraction predicates using a high-level language, proves itself helpful to model check the correctness of the algorithms. In the considered case studies, we already knew which bugs were present and we used such knowledge as a validation mechanism for testing the correct implementation of the tool (Spalazzi et al. 2018; Liva et al. 2018). In Fisher's algorithm, we also benefit from being able to test different encodings of the same mutual exclusion requirement. Indeed, once the `mutex` specification is falsified (due to not knowing that

¹⁴Bug: <https://github.com/Alluxio/alluxio/pull/7320>

two threads will never assume the same identifier), it is quite immediate to formulate an alternative specification of the same property containing a condition to eliminate spurious paths (the formula mutex_2). While the methodology is general and seems applicable in a large range of software systems, the tool implementing it is still immature, from an engineering point of view. Several extensions could be implemented to make it more usable and helpful when checking real-world software projects (e.g., inferring predicates over variables by inspecting the guards of conditional statements or loops; allowing the user to define his/her own SMT interpretations of Java data-types; . . .). At the moment, the syntax of the scripting language accepted by the tool to accomplish a software model checking task requires the user to provide detailed information about the code. On the other side, it is well known that a completely automatic tool for software model checking cannot exist. However, the process of providing this information requires less work than building a network of timed automata from Java code by hand, not considering the fact that similar engaging and repetitive tasks are error-prone when conducted by hand. More importantly, the user is driven by the tool to think at the code under analysis from a high-level abstract perspective, e.g., specifying logical predicates over variables or the number of threads in the system to be checked. The user is also helped in identifying those temporal constraints that are not written explicitly in the source code but are assumptions on the physical system that will actually run the code under analysis.

This is what we demonstrate in our experimental validation. We postpone a more detailed analysis of the applicability of the methodology and our tools with a larger number of time-dependent software projects and software developers to future work.

9 Conclusions

In this paper, we proposed a framework to extract timed automata from Java source code with temporal behaviors to formally verify time-dependent specifications. In sum, we make the following contributions:

First, the formal semantics of Java (Bogdanas and Roşu 2015) has been extended in an original way by taking temporal aspects into account.

Second, the approach that has been followed is based on the idea of extracting (by means of predicate abstraction) an abstract timed automaton for each thread in the source code. This is an improvement with respect to the related work on timed automata usually dealing with control flow abstraction. However, our framework needs more experiments with a large number of time-dependent software. An aspect that the current rules do not take into account is represented by “implicit clocks” (e.g., when the program performs a comparison between a timestamp and the current time). Intuitively, some heuristics are required to detect such situations and insert appropriate clock constraints. A precise formulation of such heuristics is part of our future work. Another aspect worth to investigate in the future is the opportunity of applying some kind of abstraction to clock variables as well (Daws and Tripakis 1998; Dierks et al. 2007; Konnov et al. 2017), thus extending the abstraction and verification framework also to recursive methods including *deadline* statements. In this respect, *counter abstractions* seem to be promising (Konnov et al. 2017).

Third, a theoretical analysis of the currently proposed extraction rules confirmed that the resulting abstraction is an over-approximation of the concrete software and, thus, preserves properties expressed in MTL or TCTL (Corollary 1). More research is required

to understand how to integrate (possibly automated) abstraction-refinement techniques to remove spurious counterexamples, and thus push further the verification task. *Finally*, the proposed framework has been presented using the Fischer’s mutual exclusion protocol as running example. This algorithm is well known to the community of timed automata, but, unlike previous works (e.g., see Salah et al. (2006)), in this work, the timed automata that model the algorithm were extracted from its Java implementation rather than manually derived from its theoretical formulation. Furthermore, the framework has been validated with two real-world Java applications, viz. Apache Kafka and Alluxio. In both cases, we were able to reproduce bugs related to a “bad” time handling, after inferring a desired timed specification for the considered threads as well as providing some interpretation of invoked libraries and of used data-structures. The main purpose of this two case studies was to show that the approach can scale to cover real-world projects and bugs. At the same time, the amount of information to be specified for driving the experiments suggests us that more work needs to be done in order to implement finer heuristic and static analysis algorithms, so that more pieces of information could be inferred automatically, without user intervention.

The interest in Java software whose behavior is time-dependent is ever greater, as witnessed by the Java Community Process (JCP) which has recently promulgated specifications for a real-time version of Java (and of the corresponding Java Virtual Machine) (Dibble and et al. 2006; Hunt and et al. 2017). Formally modeling these problems means not only having a model of the code but also a model of the scheduling algorithms. At present, the proposed framework does not take into account real-time scheduling, but, given the growing interest, we plan to address this issue in our future work.

Furthermore, slicing techniques have been proved an efficient and scalable solution for software model checking (Corbett et al. 2000). Our approach is compatible with slicing and, we believe, the integration of slicing with our tool will in future improve its scalability.

Finally, as remarked above, we have already planned to experiment our framework with a larger number of real-world Java applications and report the results in a follow-up paper.

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

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Abdulla, P.A., & Jonsson, B. (2003). Model checking of systems with many identical timed processes. *Theoretical Computer Science*, 290(1), 241–264.
- Alur, R., Courcoubetis, C., Dill, D. (1990). Model-checking for real-time systems. In: Proceedings of 5th annual IEEE symposium on logic in computer science, 1990 (LICS), IEEE, pp. 414–425.

- Aminof, B., Kotek, T., Rubin, S., Spagni, F., Veith, H. (2018). Parameterized model checking of rendezvous systems. *Distributed Computing*, 31(3), 187–222.
- Armando, A., Mantovani, J., Platania, L. (2009). Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1), 69–83.
- Ball, T., & Rajamani, S.K. (2002). The SLAM project: debugging system software via static analysis. In: ACM SIGPLAN Notices. Volume 37., ACM, pp. 1–3.
- Barrett, C., & Tinelli, C. (2018). Satisfiability modulo theories. In: Handbook of model checking. Springer, pp. 305–343.
- Barrett, C., Fontaine, P., Tinelli, C. (2017). The SMT-LIB standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa. www.SMT-LIB.org.
- Bauer, A., Leucker, M., Schallhart, C. (2011). Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4), 14.
- Beyer, D., & Keremoglu, M.E. (2011). CPAchecker: a tool for configurable software verification. In: International conference on computer aided verification. Springer, pp. 184–190.
- Beyer, D., & Wendler, P. (2012). Algorithms for software model checking: predicate abstraction vs. impact. In: Formal methods in computer-aided design (FMCAD), 2012, IEEE, pp. 106–113.
- Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R. (2007). The software model checker BLAST. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5), 505–525.
- Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R. (2009). Software model checking via large-block encoding. In: 2009 formal methods in computer-aided design. IEEE, pp. 25–32.
- Bogdanas, D., & Roşu, G. (2015). K-Java: a complete semantics of Java. *ACM SIGPLAN Notices*, 50(1), 445–456.
- Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., Larsen, K.G. (2008). Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th international workshop on java technologies for real-time and embedded systems. JTTRES '08* (pp. 106–114). New York: ACM.
- Bolléla, G., & Gosling, J. (2000). The real-time specification for Java. *Computer*, 33(6), 47–54.
- Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Ouaknine, J., Worrell, J. (2018). Model checking real-time systems. In: Handbook of model checking. Springer, 1001–1046.
- Bradley, A.R., Manna, Z., Sipma, H.B. (2006). What's decidable about arrays?. In: International workshop on verification, model checking, and abstract interpretation. Springer, pp. 427–442.
- Cimatti, A., & Griggio, A. (2012). Software model checking via IC3. In: International conference on computer aided verification. Springer, pp. 277–293.
- Cimatti, A., Griggio, A., Mover, S., Tonetta, S. (2015). Hycomp: an SMT-based model checker for hybrid systems. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp. 52–67.
- Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R. (2013). The MathSAT5 SMT solver. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp. 93–107.
- Clarke, E.M., Grumberg, O., Long, D.E. (1994). Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5), 1512–1542.
- Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H. (2000). Counterexample-guided abstraction refinement. In: International conference on computer aided verification. Springer, pp. 154–169.
- Clarke, E., Kroening, D., Sharygina, N., Yorav, K. (2005). SATABS: SAT-based predicate abstraction for ANSI-C. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp. 570–574.
- Corbett, J.C., Dwyer, M.B., Hatchiff, J., Laubach, S., Pasareanu, C.S., Robby, J.H., Zheng, H. (2000). Bandera: extracting finite-state models from Java source code. In *2000 Proceedings of international conference on software engineering (ICSE)*. IEEE (pp. 439–448).
- Cordeiro, L., Fischer, B., Marques-Silva, J. (2011). SMT-Based bounded model checking for embedded ANSI-c software. *IEEE Transactions on Software Engineering*, 38(4), 957–974.
- Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M. (2018). JBMC: A bounded model checking tool for verifying Java bytecode. In: International conference on computer aided verification. Springer, pp. 183–190.
- Cuong, N.A., & Cheng, K.S. (2008). Towards automation of LTL verification for Java pathfinder. National University of Singapore.
- Daws, C., & Tripakis, S. (1998). Model checking of real-time reachability properties using abstractions. Tools and algorithms for the construction and analysis of systems, pp. 313–329.
- De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp. 337–340.
- Dibble, P., et al. (2006). JSR 001: Real-time specification for Java (Final Release 3). From <https://jcp.org/en/jsr/detail?id=1>. (Date retrieved: October 5), 2017.

- Dibble, P. et al. (2017). The Java language specification. Java SE 9 edition From <https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf> (Date retrieved: 1st November 2018).
- Dierks, H., Kupferschmid, S., Larsen, K.G. (2007). Automatic abstraction refinement for timed automata. In: International conference on formal modeling and analysis of timed systems. Springer, pp. 114–129.
- Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A. (2015). Fairness modulo theory: a new approach to LTL software model checking. In: International conference on computer aided verification. Springer, pp. 49–66.
- Dijkstra, E.W. (1969). Notes on structured programming. *Structured programming*, 8, 1–82.
- D'silva, V., Kroening, D., Weissenbacher, G. (2008). A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7), 1165–1178.
- Dutertre, B. (2014). Yices 2.2. In: International conference on computer aided verification. Springer, pp. 737–744.
- Enderton, H.B. (1972). *A mathematical introduction to logic*. New York: Academic Press.
- Farzan, A., Chen, F., Meseguer, J., Roşu, G. (2004). Formal analysis of Java in JavaFAN. In: International conference on computer aided verification. Springer, pp. 501–505.
- Godefroid, P. (1997). Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), ACM, pp. 174–186.
- Godefroid, P. (2004). Invited talk: “model checking” software with VeriSoft. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering (PASTE)* (pp. 36–36). New York: ACM.
- Godefroid, P., Levin, M.Y., Molnar, D. (2012). Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3), 40–44.
- Grigore, R. (2017). Java generics are turing complete. In: ACM SIGPLAN notices. vol. 52. ACM, pp. 73–85.
- Gurfinkel, A., Khsai, T., Komuravelli, A., Navas, J.A. (2015). The SeaHorn verification framework. In: International conference on computer aided verification. Springer, pp. 343–361.
- Havelund, K., & Pressburger, T. (2000). Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 366–381.
- Heizmann, M., Hoenicke, J., Podelski, A. (2013). Software model checking for people who love automata. In: International conference on computer aided verification, Springer, pp 36–52.
- Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P. (1998). What's decidable about hybrid automata?. *Journal of Computer and System Sciences*, 124, 94–124.
- Herber, P., Fellmuth, J., Glesner, S. (2008). Model checking SystemC designs using timed automata. In: Proceedings of the 6th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, ACM pp. 131–136.
- Hunt, J., et al. (2017). JSR 282: Real-time specification for Java 1.1 (Early Draft Review 3). From <https://jcp.org/en/jsr/detail?id=282>, (Date retrieved: October 5, 2017).
- Jhala, R., & Majumdar, R. (2009). Software model checking. *ACM Computing Surveys (CSUR)*, 41(4), 21.
- Khsai, T., Rümmer, P., Sanchez, H., Schäf, M. (2016). JayHorn: A framework for verifying Java programs. In: International conference on computer aided verification. Springer, pp. 352–358.
- Kindermann, R., Junttila, T., Niemelä, I. (2012). SMT-based induction methods for timed systems. In: International conference on formal modeling and analysis of timed systems. Springer, pp. 171–187.
- Konnov, I., Widder, J., Spegni, F., Spalazzi, L. (2017). Accuracy of message counting abstraction in fault-tolerant distributed algorithms. In: International conference on verification, Model checking, and abstract interpretation. Springer, pp. 347–366.
- Kung, D., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y., Chen, C. (1994). On object state testing. In: Proceedings of International conference on computer software and applications conference (COMPSAC). IEEE, pp. 222–227.
- Lamport, L. (1987). A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1), 1–11.
- Landi, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4), 323–337.
- Laplante, P.A., & Ovaska, S.J. (2011). Real-time systems design and analysis: tools for the practitioner. John Wiley and Sons.
- Larsen, K.G., Pettersson, P., Yi, W. (1997). Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2), 134–152.
- Liva, G., Khan, M.T., Pinzger, M. (2017). Extracting timed automata from Java methods. In *Proceedings of the 17th IEEE international working conference on source code analysis and manipulation (SCAM)*, IEEE.

- Liva, G., Khan, M., Spegni, F., Spalazzi, L., Bollin, A., Pinzger, M. (2018). Modeling time in Java programs for automatic error detection. In: Conference on formal methods in software engineering (FMSE), Proceedings of, IEEE Computer Society, pp. 50–59.
- Luckow, K.S., Păsăreanu, C.S., Thomsen, B. (2015). Symbolic execution and timed automata model checking for timing analysis of Java real-time systems. *EURASIP Journal on Embedded Systems*, 2015(1), 2.
- Morbé, G., Pigorsch, F., Scholl, C. (2011). Fully symbolic model checking for timed automata. In: International conference on computer aided verification, Springer, 616–632.
- Nipkow, T., & Von Oheimb, D. (1998). Javalight is type-safe-definitely. In: POPL, vol. 98, pp. 161–170.
- Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V. (2009). The yogi project: software property checking via static analysis and testing. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp. 178–181.
- Păsăreanu, C.S., & Rungta, N. (2010). Symbolic pathfinder: symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM international conference on automated software engineering. ACM, pp. 179–180.
- Phan, Q.S., Malacaria, P., Pasareanu, C.S. (2015). Concurrent bounded model checking. *SIGSOFT Softw. Eng Notes*.
- Pu, G., Zhao, X., Wang, S., Qiu, Z. (2006). Towards the semantics and verification of BPEL4WS. *Electronic Notes in Theoretical Computer Science*, 151(2), 33–52.
- Rakadjiev, E., Shimosawa, T., Mine, H., Oshima, S. (2015). Parallel smt solving and concurrent symbolic execution. In: 2015 IEEE Trustcom/BigDataSE/ISPA. Volume 3., IEEE, pp. 17–26.
- Salah, R.B., Bozga, M., Maler, O. (2006). On interleaving in timed automata. In: International conference on concurrency theory. Springer, pp. 465–476.
- Schoeberl, M., Puffitsch, W., Pedersen, R.U., Huber, B. (2010). Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40(6), 507–542.
- Sen, T., & Mall, R. (2016). Extracting finite state representation of Java programs. *Software & Systems Modeling*, 15(2), 497–511.
- Spalazzi, L., & Spegni, F. (2020). Parameterized model checking of networks of timed automata with Boolean guards. *Theoretical Computer Science*. <https://doi.org/10.1016/j.tcs.2019.12.026>, <http://www.sciencedirect.com/science/article/pii/S0304397519308084>.
- Spalazzi, L., Spegni, F., Liva, G., Pinzger, M. (2018). Towards model checking security of real time Java software. In: 2018 international conference on high performance computing & simulation (HPCS), Institute of Electrical and Electronics Engineers Inc., pp. 642–649.
- Thomsen, B., Luckow, K.S., Leth, L., Bøgholm, T. (2015). From safety critical Java programs to timed process models. In: Programming languages with applications to biology and security. Springer, pp. 319–338.
- Tillmann, N., & De Halleux, J. (2008). Pex—white box test generation for .net. In: International conference on tests and proofs. Springer, pp. 134–153.
- Wang, W., & Jiao, L. (2014). Trace abstraction refinement for timed automata. In: International symposium on automated technology for verification and analysis. Springer, pp. 396–410.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



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



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



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



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



Andreas Bollin is Associate Professor at the Software Engineering Research Group of the Alpen-Adria Universität Klagenfurt, Austria. He completed his Master degree in Telematics at Graz University of Technology, where he also was a graduate assistant at the Institute for Information Systems and Computer Media. He worked for a Software consulting company (ISCN) in Dublin and Graz, before receiving his PhD in Applied Informatics at the University of Klagenfurt. His research interests include Software Comprehension and Reverse Engineering, Formal Methods, but also Didactics in Informatics and Multimedia Systems. He is member of the IEEE Computer Society and the ACM.