Towards Model-Checking Security of Real Time Java Software

Luca Spalazzi and Francesco Spegni
Università Politecnica delle Marche
Ancona, Italy

Giovanni Liva and Martin Pinzger
Alpen Adria Universität
Klagenfurt, Austria

***Abstract*—More and more software libraries and applications in high-performance computing and distributed systems are coded using the Java programming language. The correctness of such pieces of code w.r.t. a given set of security policies often depends on the correct handling of timing between concurrent or recurrent events.**

Model-checking has proven to be an effective tool for verifying correctness of software. In spite of the growing importance of this application area of formal methods, though, no one targets the problem of verifying the correctness of real-time software w.r.t. timed specifications. The few existing works focus on very different problems, such as schedulability analysis of Java tasks. We describe an approach combining rule-based static analysis together with symbolic execution of Java code to extract networks of timed automata from existing software and then use Uppaal to model-check it against timed specifications. We claim that this approach can be helpful in model-checking security policies of real time java software.

I. Introduction

High-Performance Computing (HPC), Internet of Things (IoT), Cyber-Physical Systems (CPSs), Automotive Systems are all examples of recent and growing areas where *real-time software* plays a key role. Many of them are intrinsically safety and security critical. While safety critical systems have requirements expressible in the form of invariants that are expected to hold for the entire system life, security critical systems are characterized by requirements that fall in the well known Confidentiality, Integrity, and Availability (CIA) triad. For this reasons, it is becoming crucial to have appropriate techniques and tools that guide the engineer through the task of formally verifying existing software. We underline that most of the existing approaches in literature focus on certifying that protocols or high level software models are safe and secure, but this does not imply that actual implementations of those protocols and models are exempt from bugs. For instance, in recent years two popular bugs, viz. Heartbleed [1] and GOTO-fail [2], affected popular implementation of the SSL protocol, putting several production environments and service providers at risk.

Software model checking techniques can be employed in order to certify the correctness of actual pieces of code w.r.t. given specifications, while alternative approaches such as systematic testing can only conjecture it [3]. The specifications used in software model checking can, in principle, be used to express the safety and security requirements to be checked. A prerequisite to apply software model checking, though, is to have a finite state model of the software under investigation that is as faithful as possible in describing the behavior of the original piece of code, at least w.r.t. the checked specifications. The two main research questions that drive our work are: (i) how to extract a finite-state representation of real-time Java code, and (ii) how faithful is such representation when checking safety and security policies of the software under investigation.

We present a model-checking methodology for Java software, together with a prototype implementation. The choice of Java is motivated by the fact that it is a very popular choice in two critical areas, viz. high performance computing and real-time software [4], [5] . The core components of the methodology employ rule-based approaches in order to: compute the finite-state representation of each Java thread and detect the timing constraints between events encoded in the Java program. With a case study, we show how to use the tool in order to discover a real bug that was present in a real-world project.

II. Related Work

In spite of the extensive use of Java in both high-performance computing and real-time software, only few authors (e.g. see [6], [7], [8]) focused on timed automata extraction from source code written in a general purpose programming language, as well as model checking code implementing real-time distributed algorithms. Some of them focus on the extraction of control flow automata [6], [7], not taking into account the role played by program variables along the execution. Others focus on schedulability and worst case execution time [8], [9], but do not consider the correctness w.r.t. timed specifications. To the best of our knowledge, none of them considers the problem of model-checking safety and security requirements directly from Java code. This paper is a first step towards filling the gap.

Software verification techniques can be classified [10], [11] in techniques able to work with the *concrete* state space and techniques able to work with an *abstract* state space. The term concrete indicates that such techniques are able to represent any exact change of the program variables. This approach is unfeasible whenever we have infinite or very large state programs, as often happens with software. A trade-off between time/space complexity and completeness is required. This usually means to use techniques based on *systematic execution* *exploration* [3], [12], a kind of dynamic analysis that is sound, exempt from spurious counterexamples, but incomplete since some counterexamples falsifying the checked property in the original software may not be detected [13]. An alternative is represented by *abstraction* where the concrete state space is partitioned into equivalence classes, each equivalence class being an abstract state [14], [15], [16], [17], [6], [7], [18]. Abstraction-based verification techniques are a kind of static analysis, thus they are complete but may introduce spurious counterexamples. To recap, abstraction-based verification can prove the correctness of a software whereas systematic testing can only conjecture it.

With respect to abstraction techniques, several techniques have been proposed: the most popular and effective 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. 2.c): the larger, the better, but the more complex. This means that each abstract state is denoted by a boolean combination of these predicates that over-approximates the reachable concrete states [19]. This abstraction computation is usually done using a Satisfiability Modulo Theories (SMT) solver [14], [15], [20], [21], [16], [22], [18]. Predicate abstraction can also be applied to clock variables [23]. With *control flow abstraction*, the equivalence classes are denoted by the program locations (2.d), an abstract state for each program location, i.e. a program statement [17], [6], [7], [24]. Therefore, program variables are not taken at all into account and the abstract state space coincides with the set of program locations. As a consequences, the abstract state space can be computed very quickly (no SMT solvers must be involved), but, at the cost that several program properties can not be verified. A preliminary version of this work [7] focused on Java and real-time, and thus dealt with timed automata extraction from a general purpose language, but adopted control flow abstraction and thus properties involving variable states can not be verified. This work extends that introducing predicate abstraction techniques.

III. Overview

The methodology presented in this work is based on static analysis techniques of Java code, in order to produce a network of timed automata simulating the behavior of the program. The overall task can be seen as the combination of several static analyses sub-tasks, each focusing on solving a specific sub-problem. A graphical representation is given in Fig. 1.

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), and we store it in no-sql database structure for saving time when the methodology is run interactively by the user.

A successive phase traverses the AST and along the way it *annotate timestamp variables*. Inspired by [25], [7], and using a list of common Java methods manipulating timestamps as well as Java types used to represent time values, we label as *timed variables* those variables

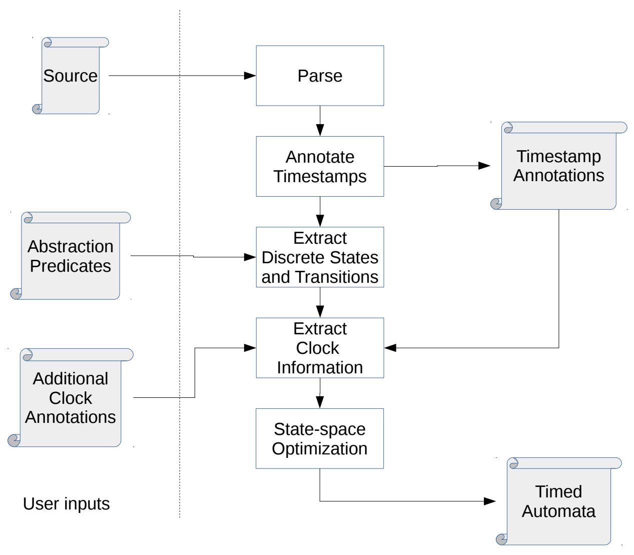

Fig. 1: A methodology for extracting networks of timed automata from Java code

in the program that are used as timestamps along the program (e.g. becuase they used to 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 the program can be maintained in a centralized way, together with the implementation of the methodology itself, and that the user is in principle allowed to add custom types and extend such sources of information. Ideally, finer implementations can allow communities of users to share their customizations of such list, to shared the acquired knowledge when analyzing Java real-time software.

The next step focuses on how to *extract discrete states and transitions* representing the program discrete behavior. To this aim, we ask 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, then we assume they can assume any value. User specified predicates are k-ary, thus they can relate at any time the concrete states of k variables. Successively, each instruction ι of the program is interpreted as a first-order predicate $\alpha(\iota)(s,t)$, relating the state of variables *before* executing the given instruction (s), to the state of the same variables *after* executing it (t). Notice that it is not possible to give a first-order interpretation of any instruction of an arbitrary Java program[1]. For this reason, this step necessarily employs a set of rules that, in principle, can be extended over time in order to detect relevant patterns appearing in Java programs. In case none of the rules applies

[1] think for example to the invocation of a recursive method on some arbitrary input

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* of the concrete instruction ι, for any ι. This is implemented by means of modern SMT solvers [] .

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

- how many *clock variables* should be added to the resulting timed automata,
- which *clock constraints* add to the discrete transitions of the timed automata, and
- what discrete transitions should *reset clock variables*.

At the moment, this stage takes advantage of the timestamp annotations added in the previous stage, together with additional so-called *clock annotations* added by the user. In the concluding section we will underline how this stage can be enhanced by means of a set of rules aiming at recognizing how timestamp variables are used along the program.

A *state-space optimization* step concludes the methodology. As will be clear in the next section, the discrete component combines the status of the *program-counter* registry, i.e. the pointer to the next instruction to be executed, together with the (abstract) state of the program variables. We can say that a (abstract) discrete state is a pair $(\alpha(\sigma), pc)$ where $\alpha(\sigma)$ is the first-order predicate abstracting the concrete variable assignment σ, while pc is the value of the program-counter registry. It is thus possible that applying some (finite) sequence of instructions $\iota_1, \ldots, \iota_n$ do not alter the abstract state of program variables $\alpha(\sigma)$, but only the program-counter component of the state. More formally, the following sequence of transitions is possible: $(\alpha(\sigma), pc) \xrightarrow{\alpha(\iota_1)} (\alpha(\sigma), pc_1) \xrightarrow{\alpha(\iota_2)} \ldots \xrightarrow{\alpha(\iota_n)} (\alpha(\sigma), pc_n)$. Under suitable conditions, such chain of states and transitions can be reduced to a single transition $(\alpha(\sigma), pc) \xrightarrow{\alpha(\iota_1);\ldots;\alpha(\iota_n)} (\alpha(\sigma), pc_n)$, eliminating the intermediate states. We will see in Sec. VI that even this simple idea proves to be very helpful in reducing the size of the extracted timed automata. This reduction can be applied whenever the specification does not look at the value of the program-counter component of the state.

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

- for model checking safety and security policies against some logical property provided by the user and exploiting some of the existing tools such as Uppaal or ... ;
- for simulation purposes, e.g. again using Uppaal or ...
- as a documentation means, giving a high-level view of the code ad our hands (e.g. for software (re-)engineering purposes).

In this work we stress the fact that the model checking purpose is very helpful in order to assist proving that pieces of real-time Java software are indeed safe and secure. Using the formalization of security properties by Clarkson and Schneider [26], [27], we claim that this approach can in principle be applied to model-check a sub-class of hyper-properties specifications, e.g. k-hyper-safety or k-liveness, i.e. safety or liveness specifications that can be falsified by systematically exploring up to k execution traces of the program under investigation.

The methodology is interactive in the sense that if the user finds the returned network of timed automata not precise enough for checking the desired security policy, he or she can provide different abstraction predicates and generate a more refined discrete component, or alternatively add more detailed clock information about the time handling of events by the program itself.

Since the parsing phase is rather standard, and techniques for labeling the timed variables has been extensively described in [25], [7], in this work we will describe how to extract the discrete and the timing component from the Java source code.

A. Attacker model

An intruder (or attacker) model is necessary when a system is model checked. Working with security protocols, the most popular intruder model is probably the *Dolev-Yao attacker* [28], [29]. The key idea behind such protocols is that all participants are expected to follow their role in the protocol under investigation, while the intruder is allowed freely allowed to alter messages in the network, send messages in any order and with any content, store messages for later use and forge new messages. The only operation is not allowed to do is to decrypt an encrypted message without the proper (symmetric or asymmetric) key. At a very abstract level, we can see the attacker as an *hostile environment* that can choose to play against the protocol participants from the outside.

In this sense, the intruder model is very similar to the Dolev-Yao one. We have to replace the *protocol* with the piece of code, the *participating actors* with the threads from which a network of timed automata is derived, and the *hostile environment* originally made of exchanged messages with the possible values assumed by the variables affecting the behavior of the threads under investigation.

Our interpretation of the *Dolev-Yao* intruder model for software is still very general and capable of modeling an *active attacker* that selects a strategy to play against the model checked piece of code. It is common practice to strengthen production software with techniques for sanitizing the input values provided by final users. Nevertheless it is also very common that programmers forget to sanitize some of the input parameters, or that the sanitation procedure is incomplete. Errors like causes vulnerabilities to remain silent in production systems for years, before being noticed. In the meantime, a smart hacker can discover it either by accident or by careful analysis of the application behavior. Such vulnerabilities may lead to invoke pieces of code after having assigned arbitrary values to variables that affect the behavior of the code itself,

```
     void MyThread(int i, int j) {
L0:    if (i >= j) {
L1:      i++;
       }
L2:    j++;
L3:    return;
     }
```

Fig. 2: A very simple source code

i.e. in a *hostile environment*. This motivates us to use this model of intruder when model checking security of Java real-time software.

Currently, the implementation of the methodology produces a network of timed automata containing the parallel composition of each thread with the intruder model. This allows to explore the behavior of each thread under any possible assignment of variables affecting the thread execution.

IV. Abstracting the discrete component

Let assume two (disjoint) finite sets of variables that we call *concrete* and *abstract*, respectively denoted by V and W. We call *abstraction* any mapping from a concrete state space onto an abstract one: $\alpha : \mathit{conf}(V) \to \mathit{conf}(W)$, where *conf* returns the set of possible configurations of the passed variable set. Let us call *program* the state transition system $P = (\mathit{conf}(V), S_0, T)$ where $S_0 \subseteq \mathit{conf}(V)$ specifies the initial states, $T \subseteq \mathit{conf}(V) \times \mathit{conf}(V)$ is the transition relation between program states induced by the program code. An abstraction α naturally induces an abstract program $\hat{P} = (\widehat{\mathit{conf}}(V), \hat{S}_0, \hat{T})$ and we call $\mathit{conf}(W) = \widehat{\mathit{conf}}(V)$.

With respect to the methodology presented in the previous section, the user inputs the program P, the set of abstract variables W and the abstraction α. The initial states $\hat{S}_0$ are obtained by composing two information: the thread *local variables* are initialized to the Java default initial values, for each variable type; the *class attributes* that affect thread execution can assume any value as initial value. Successively, an SMT-based saturation algorithm computes the abstract transition relation $\hat{T}$. It is worth underlying that if we had an explicit representation of the input program $P = (S, S_0, T, \Sigma)$, the algorithms needed for producing its abstract version would be very trivial: a simple iteration over all concrete states Σ and transitions T would suffice to build the corresponding abstract states and abstract transitions. On the other side, getting an explicit representation of a program P is, in general, unfeasible or even impossible, since P can have infinite states. We do rely, instead, on the *program text* to get the *explicit abstract* representation $\hat{P}$ from the *implicit concrete* representation of P.

Example 1. *Call P the simple program reported in Fig. 2. The program has the variables $V = \{i, j, pc\}$, where pc is the register storing the program counter values (e.g.* L0,L1,L2,L3 *in our example).*

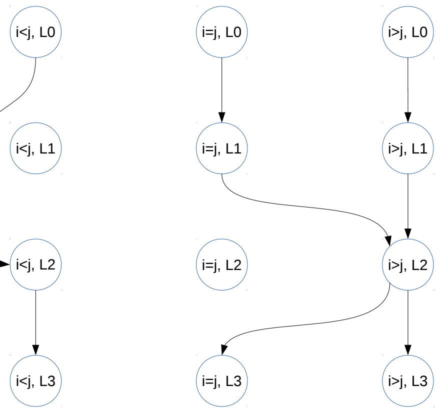

Fig. 3: Discrete component of a simple Java thread

Assume that we want to abstract the program distinguishing the cases $i < j$,$i = j$, and $i > j$, we generate a list of problems passed to the SMT solver.

Notice that the SMT solver needs not to be aware of the value of the variable pc, since the program code cannot change directly its value. Fig. 3 depicts the finite-state discrete component of the example thread computed combining the positive answers from the SMT solver together with the pc value, and then adding the corresponding (abstract) transitions. Through the SMT solver, we rule out unreachable states (e.g. $i < j$ when $pc = \mathtt{L1}$ or $i = j$ when $pc = \mathtt{L2}$) but we may introduce spurious traces (e.g. $(i = j, \mathtt{L0}) \to (i = j, \mathtt{L1}) \to (i > j, \mathtt{L2}) \to (i > j, \mathtt{L3})$), which does not correspond to any concrete execution.

Let us introduce some formal details of the adopted methodology for abstracting the discrete component of a program. Let us assume: $\mathsf{PC} ::= \mathbb{N} \mid \mathbb{N}.\mathsf{PC}$. Let us define the *increment*, *push* and *pop* operations as follows:

$$p + 1 = \begin{cases} n + 1 & \textit{if } p = n \\ n.(p' + 1) & \textit{if } p = n.p' \end{cases}$$

$$\downarrow_m p = \begin{cases} n.m & \textit{if } p = n \\ n.(\downarrow_m p') & \textit{if } p = n.p' \end{cases}$$

$$\uparrow p = \begin{cases} n & \textit{if } p = n.m \\ n. \uparrow p' & \textit{if } p = n.p' \ , p' \notin \mathbb{N} \end{cases}$$

for some $n, m \in \mathbb{N}$, $p' \in \mathsf{PC}$.

From now on, assume that both the set of concrete (resp. abstract) variables V (resp. W) contain the special variable pc tracking the program counter with domain $\mathit{dom}(pc) = \mathsf{PC}$. In order to take into account nested statements, let us assume that PC is a stack of natural numbers, with operations push, pop and sum defined on the rightmost position. For instance, let $pc = 4.2.3$, then $pc{+}1 = 4.2.4$, $\downarrow_1 pc = 4.2.3.1$ and $\uparrow pc = 4.2$. The operation $\uparrow 3$ is not defined, since the pop operation requires a program counter with at least two numbers.

Given any state s, let us write $\texttt{instr}(s)$ or $\texttt{instr}(s.pc)$ to denote the next Java statement to be executed from that states. Being Java a deterministic programming language, there is exactly one instruction associated to any value of the program counter. Assuming an asynchronous thread semantics, a program with n threads has up to n successor states from any given state, since in general the choice of the next thread to run is non-deterministic.

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 by s replacing the current value of x with z, provided that $z \in \mathit{dom}(x)$. Finally, we lift the operations on the pc variable (increment,push,pop) to the state containing it. Being a state defined by a predicate over variables, we may write a state in a first-order logical formula, denoting the corresponding first-order predicate.

Let us introduce a ternary relation: $\textsc{Next}(\hat{s}, \iota, \hat{t}) \iff \vdash \hat{s} \wedge \textsc{Post}(\hat{t}) \wedge [\iota]_{\mathrm{SMT}}$, where ι is a Java instruction assigning a value to a variable, or declaring a variable, and $\vdash \phi$ denotes that first-order predicate ϕ is satisfiable for some variable assignment. $\textsc{Post}(\hat{t})$ returns predicate $\hat{t}$ where every variable x has been replaced with a primed copy x'. The meaning of this is that we use x to denote the value of variable x in the program *before* executing a Java statement, while x' is used to denote the value of variable x in the program *after* executing the same Java statement. Finally, $[\iota]_{\mathrm{SMT}}$ computes the SMT interpretation of the Java statement ι. Notice that statement ι can be arbitrarily complex, since it can contain Java expressions and statements.

For instance, in Ex. 1, the relation $\textsc{Next}((i < j, \texttt{L0}), \texttt{i++}, (i < j, \texttt{L1}))$ holds because $\vdash (i < j) \wedge (i' < j') \wedge (i' = i + 1 \wedge j' = j)$ holds (e.g. taking $i = 1$ and $j = 3$). On the other side, $\textsc{Next}((i < j, \texttt{L0}), \texttt{i++}, (i > j, \texttt{L1}))$ does not hold, because for all concrete values of i and j: $\nvdash (i < j) \wedge (i' > j') \wedge (i' = i + 1 \wedge j' = j)$. Notice that the transformation applied by any Java instruction ι does not depend on the value of the program-counter, but only depends on the configuration of the program variables. For this reason the program-counter component in the abstract state can be ignored, when generating the SMT problem to solve.

Given a finite set of transitions X, we write $\mathit{final}(X)$ to denote the set of final states reachable with transitions in the set: $\{t \mid \exists s.(s,t) \in X, \forall s'.(t,s') \notin X\}$.

Given an abstract state $\hat{s}$ and an instruction ι, we can compute the set of transitions departing from s when applying ι by means of several $\textsc{Reach}$ operators, one for each syntactic category of the instructions provided by the Java language. At the moment our implementation covers only a subset of the Java 8 language, focusing on the core control structures, viz. *assignments*, *sequential execution*, *method-calls*, *if-then-else*, *while*, *for*, *for-each*. Due to lack of space, we show in Tab. I an excerpt of the formally defined and implemented $\textsc{Reach}$ operators.

Notice that the $\textsc{Reach}_{\texttt{ITE}}$ operator allows in principle, from a same configuration, to reach some states in the *then*-branch (i.e. transition $(s, \downarrow_0\downarrow_0 s)$), and in the *else*-branch (i.e. transition $(s, \downarrow_0\downarrow_1 s)$). The same can happen with $\textsc{Reach}_{\texttt{While}}$, where both the guard and its negation can be satisfiable, starting from the same abstract configuration. This is consistent with the existential nature of the abstraction.

Notice also that guard γ 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 γ to a sequence of (intermediate) variable assignments, and next a (functionally) equivalent γ' guard without side-effects replaces γ.

Finally, $\textsc{Reach}_{\top}$ is the rule catching every instruction for which a first-order interpretation cannot be given. This rule is invoked also when interpreting an expression or a statement that would fall in one of the handled syntactic categories, but one of the variables affecting the execution of such statement is not in the set of abstract variables W. The ratio behind this choice is that since we don't know the value of such variable, the expression containing it can return any possible value. This is an example of how the effects of our intruder model are encoded in the computed discrete component.

Given a single Java thread P and an abstraction α, we can now define what it means to compute its (abstract) transition relation, given the abstract state space $\mathit{conf}(W)$ induced by the abstraction α is the codomain of mapping α:

$$\textsc{Reach}(P, \alpha) = \bigcup_{s \in \mathit{conf}(W)} \textsc{Reach}_{[P]}(s, P)$$

Alg. 1 shows the procedure for obtaining a finite-state automaton describing the discrete behavior of a single Java thread.

Algorithm 1

```
function $\textsc{Thread2FSA}(P, S_0, \alpha)$
    $\hat{T} = \textsc{Reach}(P, \alpha)$
    $\hat{S}_0 = \alpha(S_0)$
    $\hat{S} = \hat{S}_0 \cup \{t \mid \exists s \in \hat{S}.\ (s,t) \in \hat{T}\}$
    $\hat{P} := (\hat{S}, \hat{S}_0, \hat{T})$
    return $\hat{P}$
```

Lemma 1. *Algorithm 1 always terminates.*

Proof. We begin by observing that the $\textsc{Reach}$ operators are recursively defined. Any sequence of recursive calls to $\textsc{Reach}$, though, reduces the size of the statement to be processed, with the only exception of $\textsc{Reach}_{\texttt{While}}$. If no $\textsc{Reach}_{\texttt{While}}$ happens in the sequence, then the sequence of recursive calls is obviously finite.

In case a $\textsc{Reach}_{\texttt{While}}$ occurs in the sequence, we observe that the set of transitions produced at each step by $\textsc{Reach}_{\texttt{While}}$ is monotone 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 $\mathit{conf}(W) \times \mathit{conf}(W)$, which is finite due to the employed abstraction). This implies the statement. $\square$

V. Abstracting the timing of events

To model the passage of time, we add a finite set of clock variables C to the model. We call Γ the set of guards on

$$
\begin{aligned}
\textsc{Reach}_{\texttt{Assign}}(s,\iota) ={}& \{(s,t) \mid t \in \mathit{conf}(W), t.\mathit{pc} = s.\mathit{pc}+1, \textsc{Next}(s,\iota,t)\} \\
& \hfill \textit{when } \iota = \texttt{instr}(s) \\
\textsc{Reach}_{\texttt{Seq}}(s,\iota) ={}& X \cup \textstyle\bigcup_{t \in \mathit{final}(X)} \textsc{Reach}_{[B']}(t, B') \\
& \hfill \textit{when } \iota = \texttt{instr}(s), \textit{and } \iota = B;B' \\
\textsc{Reach}_{\texttt{ITE}}(s,\iota) \supseteq{}& \{(s, \downarrow_0\downarrow_0 s)\} \cup X \cup \{(t, (\uparrow\uparrow t)+1) \mid t \in \mathit{final}(X)\} \\
& \hfill \textit{when } \iota = \texttt{instr}(s), \vdash s \wedge \gamma, X = \textsc{Reach}_{[B]}(\downarrow_0\downarrow_0 s, B), \\
& \hfill \textit{and } \iota = \text{if } (\gamma)\ \{\ B\ \} \text{ else } \{\ B'\ \} \\
\textsc{Reach}_{\texttt{ITE}}(s,\iota) \supseteq{}& \{(s, \downarrow_0\downarrow_1 s)\} \cup X \cup \{(t, (\uparrow\uparrow t)+1) \mid t \in \mathit{final}(X)\} \\
& \hfill \textit{when } \iota = \texttt{instr}(s), \vdash s \wedge \neg\gamma, X = \textsc{Reach}_{[B']}(\downarrow_0\downarrow_1 s, \text{B'}), \\
& \hfill \textit{and } \iota = \text{if } (\gamma)\ \{\ B\ \} \text{ else } \{\ B'\ \} \\
\textsc{Reach}_{\texttt{While}}(s,\iota) \supseteq{}& \{(s, \downarrow_0 s)\} \cup X \cup \textstyle\bigcup_{t \in \mathit{final}(X)} \textsc{Reach}_{\texttt{While}}(\uparrow t, \iota) \\
& \hfill \textit{when } \iota = \texttt{instr}(s), \vdash s \wedge \gamma, X = \textsc{Reach}_{[B]}(\downarrow_0 s, B), \\
& \hfill \textit{and } \iota = \text{while } (\gamma)\ \{\ B\ \} \\
\textsc{Reach}_{\texttt{While}}(s,\iota) \supseteq{}& \{(s,t) \mid t = s+1\} \\
& \hfill \textit{when } \iota = \texttt{instr}(s), \vdash s \wedge \neg\gamma, \\
& \hfill \textit{and } \iota = \text{while } (\gamma)\ \{\ B\ \} \\
\textsc{Reach}_{\top}(s,\iota) ={}& \mathit{conf}(W) \\
& \hfill \textit{when } \iota = \texttt{instr}(s), \textit{and } [\iota]_{\text{SMT}} \textit{ is not defined}
\end{aligned}
$$

TABLE I: Subset of rules for extracting the discrete component from a Java program

clock variables, comparing clocks among themselves or with constants. A *timed automaton* is a state transition system $P = (S, S_0, T, C, I, G, R)$ such that (S, S_0, T) is a finite-state automaton, $I : S \to \Gamma$ maps each discrete state to its time invariant, $G : T \to \Gamma$ maps each discrete transition to one (possibly universal) enabling clock guard, and $R : T \to 2^C$ maps each discrete transition to zero or more clock variables to reset when taking the transition. Let us underline that Java programs don't handle natively clock variables. They keep track of time passage by means of timestamps, and along the program the latter are compare against some view of the system clock. We assume that each thread has a clock $c(t_1, t_2) \in C$ for each pair of timestamp variables (t_1, t_2) found by the "Annotate Timestamps" stage (cfr. Fig 1). If the user manually specifies additional timestamp variables, more clock variables will be generated. Intuitively, every clock $c(t_1, t_2)$ is used to keep track of the difference between the timestamps. Thus, every clock condition of the form $c(t_1, t_2) \sim k$ holds iff $t_1 - t_2 \sim k$, for any comparison operator $\sim \in \{<, \leq, =\}$ and constant $k \in \mathbb{N}$.

At the moment we decided to handle threads where timestamp variables fall in one of the following classes:

- **now-timestamp**: these are the timestamp variables that are updated only for storing the value of the system clock,
- **constant-timestamp**: these are the timestamps that are assigned only once along the life of the thread.

Restricting our analysis to such timestamps, helps us in recognizing the encoded time constraints. First of all, we only map a clock variable $c(t_1, t_2)$ to timestamps t_1 and t_2 only if t_1 is a now-timestamp and t_2 is a constant-timestamp, thus reducing the number of clock variables in the network of timed automata. This equals to assigning a clock variable $c(t_1)$ that only depends on the now-timestamp, and make t_2 a parameter of the timed automaton appearing in the clock expressions. Indeed, for any constant $k \in \mathbb{N}$, $c(t_1, t_2) \sim k$ iff $t_1 - t_2 \sim k$ (by definition) iff $c(t_1) \sim t_2 + k$, and by our assumption the expression $t_2 + k \in \mathbb{N}$ is constant as well. As a consequence, every time in the code appears the expression $t_1 \sim t_2$ guarding some code block, we must add the clock constraint $c(t_1) \sim t_2$ in order to enter the abstraction of the code block, and clock constraint $\neg(c(t_1) \sim t_2)$ in order to skip it.

From the point of view of security analysis, notice that flagging t_2 as a parameter of the timed automaton means to give one more chance to the intruder to create a *hostile environment* where the piece of code could be executed. This in turn implies that we need to model-check all the possible (abstract) values that parameter t_2 may assume.

Note that when updating a timestamp in the Java code, the configuration of the discrete component does not change. On the other side, our methodology must understand when a clock variable $c(t_1, t_2)$ should be reset. Under our assumptions, since the only timestamp to be updated is t_1 and it is used for checking the current real time, it is enough to reset clock $c(t_1)$ the first time variable t_1 reads the system clock, and the successive times no clock reset must happen.

VI. Case Study

We have implemented in a prototype tool the interactive abstraction and verification methodology presented in this work. In the tool the user specifies the program he/she wants to abstract and a set of predicates over program variables. Here we share some preliminary result, in the form of a case study conducted trying to identify a bug that was present in a real-time Java project. The method in Fig. 4 constitutes the body of a Java thread part of a project named Apache Kafka[2], a popular distributed streaming platform allowing to implement a publish-subscribe service to streams of data, and process streams of data with an eye to fault-tolerance, as usual for distributed algorithms.

The considered piece of code implements a poll mechanism, where a server is periodically checked for being ready, and if not, it is enforced to become ready. That method contained a bug[3] appearing when the parameter `timeout` assumes a negative value, or a big enough value, such that expression

[2]Source code: `https://github.com/apache/kafka`

[3]Bug: `https://issues.apache.org/jira/browse/KAFKA-4290`

```
public void poll(long timeout) {
  // poll for io 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. 4: Example of real-time Java code with security bug

`now + timeout` overflows and becomes smaller than `now`. The bug was found "out of the box", by the programmers' own admission, and we claim that our methodology could have helped in finding it if employed in a systematic investigation of methods and their correctness and security specifications.

In our case study, the thread security requirement is an *availability* requirement, that can be described as follows: whenever the thread exits, the server must be in its *ready* state, denoting that the service is available to the other threads in the system.

To this aim, the user instruct the tool to abstract the add abstract variables `is_ready` and `coordinator_known` to the system. 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 a special SMT variable used to store the result of invoking the method invocation. All other methods are abstracted with a first-order tautology, meaning that they have no effect on variables `is_ready` and `coordinator_known`. Finally, the user specifies that he/she wants to verify a system with only the `poll` thread. The tool automatically recognizes two timestamps, viz. `deadline` and `now`, the former being a *constant-timestamp* and the latter being a *now-timestamp*.

Fig. 5 reports a graphical representation of the timed automaton corresponding extracted from thread `poll`. Note that the actual state space of the discrete component is given by 204 states, i.e. 4 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`. In the figure we only make explicit the value of the program-counter register,

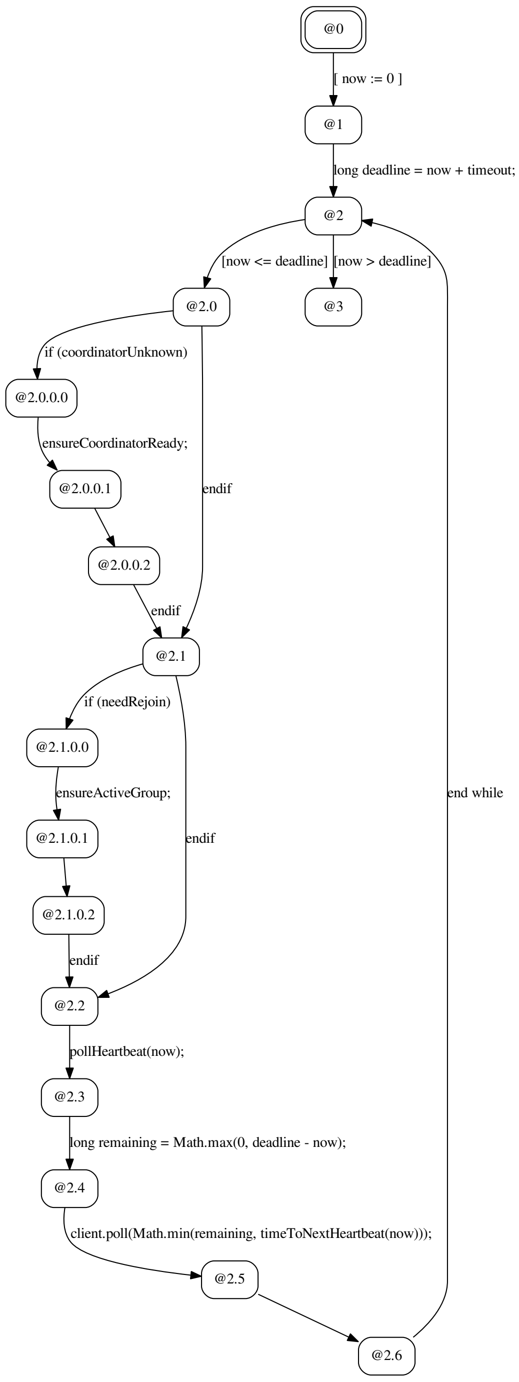

Fig. 5: The extracted timed automaton

for the sake of readability. The timed automaton contains one clock variable `now`, and one parameter `deadline`.

After the parse stage, the tool takes 7.19 seconds to generate a Uppaal representation of the timed automaton from the `poll` thread on an Intel i7 machine with 8 cores and 16GB of RAM. For solving the SMT problems we used the Z3 SMT solver, that was invoked 204 times. The security requirement can be encoded with the following TCTL formula: `AF (is_ready = true)`. Uppaal takes less than a second to find the following counter example path: $(\sigma, pc = @0) \rightarrow (\sigma, pc = @1) \rightarrow (\sigma, pc = @2) \rightarrow (\sigma, pc = @3)$, where $\sigma = deadline < 0 \wedge is\_ready = false \wedge coordinator\_known = true$, and a simple code inspection allows to understand that the found counterexample is not spurious, i.e. is not added by the abstraction process, but it can happen with concrete executions of the method.

VII. Conclusions

This paper describes a methodology aiming at model checking some safety and security requirements against pieces of Java code. The methodology can extract a network of timed automata model of the Java code under analysis, and through an interactive workflow, can accept information from the user that drives the process of generating an existential abstraction of the checked code. The process of extracting a model from Java code is cumbersome and error-prone, thus we claim that our methodology can be helpful if plugged into the usual software engineering and re-engineering processes, to prove that existing code fulfills a set of safety and security specifications, provided that the latter can be expressed using the temporal logic TCTL. The presented methodology has been implemented in a prototype tool. The tool implements several steps of static analysis, extracting a piece of information at each step.

Our work is original in several aspects: first, similar works extracting timed automata from code deal with control flow abstraction [25], [7], [6] and largely overapproximate the visited program states, which in turns means the model-checking phase is not successful because of the great number of spurious counter-examples introduced. Allowing the user to decide which variable configurations to track, he/she is in power to find the best trade-off between model refinement and feasibility of model-checking. Second, existing approaches for model-checking real-time Java code only focus on the analysis on best-/worst-case execution times, not considering the problem of checking the *correctness* of the software w.r.t. temporal, safety or security requirements. We aim at filling this gap. The approach we propose needs more experimental validation and must be tested for scalability when analyzing complex scenarios with several threads, sharing variable or synchronizing with the mechanisms provided by the Java language. Also, model checking more complex safety and security specifications should be tested. Finally, techniques for abstracting clocks from source code are needed, since existing ones do not apply to this context [23], [30].

References

[1] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[2] M. Bland, "Finding more than one worm in the apple," *Queue*, vol. 12, no. 5, p. 10, 2014.

[3] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997, pp. 174–186.

[4] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the high performance computing arena: Research, practice and experience," *Science of Computer Programming*, vol. 78, no. 5, pp. 425–444, 2013.

[5] S. Ekanayake, S. Kamburugamuve, and G. C. Fox, "Spidal java: high performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium*. Society for Computer Simulation International, 2016, p. 3.

[6] P. Herber, J. Fellmuth, and S. Glesner, "Model checking systemc designs using timed automata," in *Proc. of the 6th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. ACM, 2008, pp. 131–136.

[7] G. Liva, M. T. Khan, and M. Pinzger, "Extracting timed automata from java methods," in *Proceedings of the 17th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2017.

[8] K. S. Luckow, C. S. Păsăreanu, and B. Thomsen, "Symbolic execution and timed automata model checking for timing analysis of java real-time systems," *EURASIP Journal on Embedded Systems*, vol. 2015, no. 1, p. 2, 2015.

[9] B. Thomsen, K. S. Luckow, L. Leth, and T. BØgholm, "From safety critical java programs to timed process models," in *Programming Languages with Applications to Biology and Security*. Springer, 2015, pp. 319–338.

[10] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[11] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

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

[13] P. Godefroid, "Invited talk: "model checking" software with verisoft," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '04. New York, NY, USA: ACM, 2004, pp. 36–36. [Online]. Available: http://doi.acm.org/10.1145/996821.996824

[14] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 1–3.

[15] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5, pp. 505–525, 2007.

[16] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, J. H. Robby, and H. Zheng, "Bandera: Extracting finite-state models from java source code," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*. IEEE, 2000, pp. 439–448.

[17] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata," in *Int. Conf. on Computer Aided Verification*. Springer, 2013, pp. 36–52.

[18] T. Sen and R. Mall, "Extracting finite state representation of java programs," *Software & Systems Modeling*, vol. 15, no. 2, pp. 497–511, 2016.

[19] D. Beyer and P. Wendler, "Algorithms for software model checking: Predicate abstraction vs. impact," in *Formal Methods in Computer-Aided Design (FMCAD), 2012*. IEEE, 2012, pp. 106–113.

[20] D. Beyer and M. E. Keremoglu, "A tool for configurable software verification," in *CAV. LNCS*, vol. 6806. Springer, 2011, pp. 184–190.

[21] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Satabs: Sat-based predicate abstraction for ansi-c," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 570–574.

[22] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On object state testing," in *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*. IEEE, 1994, pp. 222–227.

[23] H. Dierks, S. Kupferschmid, and K. G. Larsen, "Automatic abstraction refinement for timed automata," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2007, pp. 114–129.

[24] G. Pu, X. Zhao, S. Wang, and Z. Qiu, "Towards the semantics and verification of bpel4ws," *Electronic Notes in Theoretical Computer Science*, vol. 151, no. 2, pp. 33–52, 2006.

[25] G. Liva, M. T. Khan, F. Spegni, L. Spalazzi, A. Bollin, and M. Pinzger, "Modeling time in java programs for automatic error detection," in *Conference on Formal Methods in Software Engineering, Proceedings of*, 2018.

[26] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[27] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," in *International Conference on Principles of Security and Trust*. Springer, 2014, pp. 265–284.

[28] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Information Theory*, vol. 29, no. 2, pp. 198–207, 1983. [Online]. Available: https://doi.org/10.1109/TIT.1983.1056650

[29] C. A. Meadows, "Formal verification of cryptographic protocols: A survey," in *International Conference on the Theory and Application of Cryptology*. Springer, 1994, pp. 133–150.

[30] I. Konnov, J. Widder, F. Spegni, and L. Spalazzi, "Accuracy of message counting abstraction in fault-tolerant distributed algorithms," in *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*. Springer, 2017, pp. 347–366.