PETRIFY: Petri-net Based Analysis of Concurrency Properties in Java Bytecode

Akshatha Shenoy[1] [0009-0004-2439-1656] and Carlo A. Furia[1] [0000-0003-1040-3201]

Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland
shenoa@usi.ch bugcounting.net

Abstract. The landscape of automated formal verification is populated by techniques that make prominently different trade-offs: some focus on expressiveness and precision, supporting the verification of complex properties; others favor scalability and practicality, so that they are applicable to larger programs using different features. This paper presents PETRIFY, a novel automated verification technique for concurrency properties that achieves a distinctive trade-off. PETRIFY encodes the semantics of Java bytecode programs into Petri nets (PNs), which can be analyzed by state-of-the-art model checking tools such as LoLA. As our experiments demonstrate, PETRIFY's approach offers an interesting combination of expressiveness and practicality: PNs are a fairly precise encoding of the concurrent behavior of programs; at the same time, PETRIFY's PN encoding is succinct, so that its analysis remains quite insensitive to parameter size. Another practical benefit of targeting bytecode is that jPETRIFY, the prototype tool that implements the PETRIFY technique, is applicable, in principle, to programs written in any version of Java and even a subset of Kotlin (another language that compiles to Java bytecode) while other similar tools are limited to older versions of Java. While this paper's experiments focus on analyzing fundamental properties like deadlock, PETRIFY's approach lends itself to be extended to other kinds of concurrency analysis, which we plan to tackle in future work.

1 Introduction

Every automated formal program analysis technique has to contend with the expressiveness vs. scalability trade-off. Techniques that favor expressiveness support precisely verifying all sorts of user-defined properties, but may struggle to analyze large programs. In contrast, techniques that target scalability are based on bespoke abstractions that approximate only specific hardcoded properties, but are applicable to realistic-size programs. In the domain of concurrent verification, for example, software model checkers (e.g., Java Pathfinder [10]) favor expressiveness as they support temporal logic specifications and other kinds of program annotations, and can analyze path conditions precisely. In contrast, custom static analyzers that focus on a single property (e.g., data races for Infer's RacerD [1] or deadlocks for JaDA [19]) are usually more efficient but cannot verify other kinds of properties and may suffer from imprecision. Another important dimension of practicality for any formal verification tool is *language support*. Most analysis techniques work at the source-code level, which entails that they may struggle to keep up with the evolution of modern languages. Consider, again, the example of Java: tools like Pathfinder and JaDA were developed for earlier versions of Java, and hence

cannot analyze programs that include recently introduced features—even if the features themselves do not affect the concurrent behavior. All the more so, applying a verification technique to work on a (subset of a) different language (even one that is similar to Java) is usually onerous—in terms of both adapting the technique and developing a suitable implementation toolchain.

This paper presents PETRIFY, a program analysis technique for concurrent Java programs that explores a novel trade-off between expressiveness, scalability, and language support. PETRIFY relies on two key ideas. First, it encodes the concurrent behavior of a Java program as a Petri net. Thanks to recent advances in their algorithmic verification, PNs have become an appealing abstract model of concurrent computation, which combines a high expressiveness with highly optimized model-checking tools. Second, PETRIFY translates directly from Java bytecode, rather than working at the source code level. This makes it a technique that is not tied too closely to a specific version of Java, and even works for programs written in (an imperative subset of) Kotlin—another language that compiles to Java bytecode—which provides an additional element of flexibility and practicality.

In our experiments to demonstrate PETRIFY's capabilities, we implemented it in a tool called jPETRIFY, and applied it to detect deadlocks in 36 Java and 3 Kotlin programs taken or adapted from various benchmarks. Although our prototype implementation of jPETRIFY currently only supports a limited number of properties out of the box, the experiments demonstrated some of its strengths, which complement other state-of-the-art tools such as Java Pathfinder and JaDA. In particular, jPETRIFY is applicable to Java programs written in any version of the language, whereas other tools usually cannot process versions more recent than Java 11. In addition, PETRIFY's abstractions are insensitive to the parameter size, so that jPETRIFY scales up to examples that heavier, more precise tools cannot handle.

Contributions. The paper makes the following main contributions: *i*) PETRIFY: a novel technique to analyze concurrency properties of bytecode programs based on a flow- and context-sensitive, path-insensitive encoding of bytecode programs into Petri nets. *ii*) jPETRIFY: an implementation of PETRIFY based on the Soot static analyzer and the LoLA Petri net model checker. *iii*) An experimental evaluation of jPETRIFY on 39 programs. *iv*) The implementation of jPETRIFY and the experimental artifacts are available [27].

2 An Overview of PETRIFY

Fig. 1a shows a simple concurrent Java program with two threads `thread_12` and `thread_21` that try to acquire a lock on variables `lock1` and `lock2`. It is easy to see that the program will deadlock if the two threads *interleave* their lock acquisition operations, leading to a state where `t_12` has a lock on `lock1`, `t_21` has a lock on `lock2`, and each thread waits for the other thread to release the lock it is holding. Let's describe how PETRIFY analyzes this program to find the deadlock scenario.

PETRIFY works on *bytecode* (produced by the Java compiler), rather than on source Java code. One advantage of targeting bytecode is that we can support any Java version, since new language features (e.g., Fig. 1a's instance main method and unnamed class, which were introduced in Java 21 and still are preview features) are desugared into a stable set of bytecode instructions by the compiler.

PETRIFY first translates the input bytecode program into the RB program shown in Fig. 1b. RB is an intermediate language that is flow- and context-sensitive but path-

```
1  void main() {
2      Object lock1 = new Object();
3      Object lock2 = new Object();
4      Runnable r_12 = () -> { synchronized (lock1) { synchronized (lock2) {}}};
5      Runnable r_21 = () -> { synchronized (lock2) { synchronized (lock1) {}}};
6      Thread t_12 = new Thread(r_12);
7      Thread t_21 = new Thread(r_21);
8      t_12.start();
9      t_21.start();
10 }
```

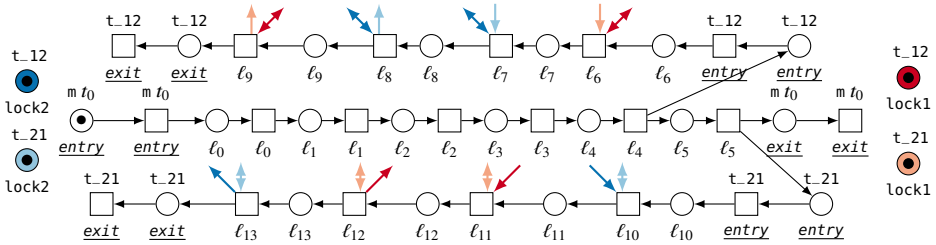
(a) A two-threaded Java program that may deadlock.

```
                 def main
entry_main:      begin
          ℓ0:    write lock1
          ℓ1:    write lock2
          ℓ2:    write t_12
          ℓ3:    write t_21
          ℓ4:    fork r_12 t_12
          ℓ5:    fork r_21 t_21
exit_main:       end

                 def r_12
entry_r_12:      begin
          ℓ6:    acquire lock1
          ℓ7:    acquire lock2
          ℓ8:    release lock2
          ℓ9:    release lock1
exit_r_12:       end

                 def r_21
entry_r_21:      begin
         ℓ10:    acquire lock2
         ℓ11:    acquire lock1
         ℓ12:    release lock1
         ℓ13:    release lock2
exit_r_21:       end
```

(b) Petrify's translation of Fig. 1a's program into the rb intermediate language.



(c) Petrify's encoding of Fig. 1b's rb program as a Petri net. To reduce clutter, the arrows connecting transitions $\ell_{6\text{–}9}$ and $\ell_{10\text{–}13}$ to the four places ●, ●, ●, ● are shortened and rely on colors to indicate the connected place.

Fig. 1: A simple concurrent Java program, and Petrify's encoding in rb and as a Petri net.

insensitive; hence, it represents an approximationof the bytecode program's executions. We introduced rb to reduce the semantic gap between bytecode and Petri nets, which simplifies the design of the overall translation and also reasoning about its correctness. In this example, rb captures all aspects of concurrent behavior without information loss.

Then, Petrify encodes the semantics of Fig. 1b's rb program into the Petri net (PN) in Fig. 1c. Petrify supports the PN format used by state-of-the-art analyzers such as LoLA [31], so that its output can be fed to these tools to analyze any properties of interest. To check the presence of *deadlocks*, Petrify augments the output PN with additional "monitoring" places and transitions (now shown in Fig. 1c for simplicity) and a suitable temporal logic formula. With this input, LoLA quickly finds a PN execution that corresponds to Fig. 1a's deadlock.

As you can glean from Fig. 1c, the structure of the PN is clearly modular. The central chain of nodes (circular "places" and square "transition" in PN terminology) encodes the control flow of the `main` function, whereas the top and bottom chains correspond to the anonymous methods (lambdas) executed by threads `t_12` and `t_21` respectively. The arrows connecting transitions ℓ_4 and ℓ_5 to the entry places of `t_12` and `t_21` represent the starting points of the threads parallel to `main`. The modular structure of Petrify's PN

encoding also helps map back an error trace given by the PN analyzer to an execution of the original Java program.

3 Related Work

There is a vast amount of research on detecting *concurrency* programming errors. For space constraints, we focus on techniques and tools that are applicable to Java and are currently available; as for any kind of program analysis, they can be broadly classified in static or dynamic, and according to their expressiveness vs. scalability.

Static techniques are usually sound but imprecise. *Deductive* verification offers high expressiveness, as it can analyze user-defined complex properties; however, it requires a significant amount of human effort, as programs must be annotated with detailed formal specifications and additional assertions such as invariants. VerCors [2] and VeriFast [11] are two prominent examples of deductive verifiers for Java, both supporting annotations written in a combination of JML [20] and separation logic. VerCors offers a higher degree of automation, as it relies on the Viper intermediate verifier [26] to discharge verification conditions, while VeriFast supports interactive correctness proofs.

Model checking is a popular verification technique based on analyzing (finite-)state models against temporal-logic properties. Tools like JPF (Java Pathfinder) [10], JayHorn [14], JBMC [5], and JMC [13] are *software* model checkers: they encode the semantics of a Java program using a state model that can be analyzed by a model checker such as Spin [30]. This provides a high degree of automation, while still supporting user-defined properties (usually expressible in temporal logic). Petrify follows a similar approach, but it leverages Petri nets (instead of less expressive automata models) to naturally model several aspects of a program's concurrent behavior.

A key issue when designing an analysis based on model checking is that the full semantics of a (Java) program is generally infinite-state. Some tools (e.g., JPF, JBMC, JMC) perform a *bounded* exploration of the infinite program state; hence, they are powerful testing tools, but are not sound in general. JPF, in particular, is a mature analysis framework based on a custom controllable version of the Java virtual machine; it can systematically or randomly explore execution paths and different thread interleavings. Symbolic JPF (SPF) performs the state exploration *symbolically* (using a form of symbolic execution), so that it is systematic and satisfies complex coverage criteria.

Other techniques build a finitely-analyzable *over-approximation* of a program's state space, which loses precision but retains soundness. Techniques like Infer's RacerD [6,1] (based on separation logic and bi-abduction), Checkmate [8] (based on abstract interpretation) and Chord [15] (an unsound, precise technique based on context-sensitive analyses) are all different applications of static analysis to detect data races. JaDA [19] offers deadlock detection for (a subset of) Java bytecode; it uses typing rules to define an infinite-state abstract model of the program's lock dependencies; the model is analyzable by means of a fixpoint decision algorithm. Petrify also builds, by means of a dataflow analysis, an approximation of a Java program's executions, which it encodes as a finitely-analyzable Petri net.

Petri nets are a classic model of concurrency. To our knowledge, they have been historically mainly used to build abstract, high-level models, to encode the semantics of core concurrency properties (e.g., causal atomicity [7]) and primitives (e.g., synchronization

$$
\begin{aligned}
\mathit{Jimple} &::= \mathit{Method}^* \\
\mathit{Method} &::= \texttt{method}\ \mathit{Id}\colon \mathit{Type}^* \to \mathit{Type}\ \{\mathit{Instr}^*\} \\
\mathit{Instr} &::= \texttt{noop} \mid \texttt{goto}\ \mathit{Label} \mid \texttt{if}\ \mathit{Var}\ \mathit{Label} \mid \\
&\quad \texttt{switch}\ \mathit{Var}\ (\mathit{Var}\colon \mathit{Label})^* \mid \\
&\quad \mathit{Var} := \mathit{Expr} \mid \texttt{return}\ \mathit{Var} \mid \texttt{return} \mid \\
&\quad \texttt{monitor_enter}\ \mathit{Var} \mid \texttt{monitor_exit}\ \mathit{Var} \\
\mathit{Expr} &::= \mathit{Var} \oplus \mathit{Var} \mid \odot \mathit{Var} \mid \mathit{Const} \mid \mathit{Call} \\
\mathit{Call} &::= \texttt{invoke}\ \mathit{Id}\ \mathit{Var}^*
\end{aligned}
$$

(a) Syntax of Jimple.

$$
\begin{aligned}
\mathit{RB} &::= \mathit{Proc}^* \\
\mathit{Proc} &::= \texttt{def}\ \mathit{Id}\ \texttt{begin}\ \mathit{Cmd}^*\ \texttt{end} \\
\mathit{Cmd} &::= \texttt{skip} \mid \\
&\quad \texttt{goto}\ \mathit{Label} \mid \texttt{jump}\ \mathit{Label}\ \mathit{Label} \mid \\
&\quad \texttt{read}\ \mathit{Var} \mid \texttt{write}\ \mathit{Var} \mid \\
&\quad \texttt{acquire}\ \mathit{Var} \mid \texttt{release}\ \mathit{Var} \mid \\
&\quad \texttt{fork}\ \mathit{Id}\ \mathit{Thread} \mid \texttt{join}\ \mathit{Thread} \mid \\
&\quad \texttt{call}\ \mathit{Id}
\end{aligned}
$$

(b) Syntax of RB.

Fig. 2: Syntax of the intermediate representations used by Petrify.

through conditional variables [4]). In our work, we leverage the recent advances in model-checking tools for Petri nets [3,31,16] and use them as back-end of jPetrify.

Dynamic analysis has become more popular in recent years to detect concurrency bugs such as deadlocks [29], data races [28,18], and linearizability [17,21] and atomicity [25,24] violations. While such techniques do not offer soundness, they are practical (i.e., since they are based on executing a program, they support all language features) and scalable (e.g., they analyze a trace in linear time).

Soundness vs. precision. On paper, static and dynamic analysis offer complementary advantages and disadvantages: static techniques overapproximate program behavior, and hence their results are sound (exhaustive) but imprecise; dynamic techniques underapproximate program behavior, and hence their results are precise (no false alarms) but unsound. In practice, the boundary between soundness and precision is somewhat fuzzy, and even static techniques are very often unsound in certain cases [23], because they may not fully support certain language features due to practical concerns. As we will discuss in Sec. 5, Petrify follows a similar approach of being "mostly" sound and "reasonably" precise: *i*) Petrify's implementation does not (fully) support certain Java language features; thus, the analysis results may be unsound for programs that use those features. *ii*) Petrify relies on an alias analysis to identify possibly shared lock and thread variables; when the alias analysis results are insufficiently precise, Petrify's own analysis may in turn become unsound or imprecise. Ultimately, the practicality of Petrify in analyzing concurrent programs is established with Sec. 6's experimental evaluation, which highlights its capabilities and limitations, also in comparison to other similar concurrency analysis tools.

4 Preliminaries: Jimple Bytecode, RB, and Petri nets

Petrify encodes the concurrent dataflow semantics of a JVM bytecode program as a Petri net. This section introduces the intermediate representations used by Petrify.

4.1 The Jimple Bytecode Representation

Rather than working directly with JVM bytecode, Petrify uses Soot's Jimple bytecode representation: a typed bytecode form that abstracts several low-level details and incorporates static information. Fig. 2a outlines the main constructs of Jimple, using a simplified abstract syntax. Before going through it, we point out that Jimple uses an SSA (static

single assignment) form; thus, any complex expression $((v_0 \oplus_1 v_1) \oplus_2 v_2) \cdots \oplus_n v_n)$ becomes a sequence of assignments to local variables $r_1 := v_0 \oplus_1 v_1$, $r_2 := r_1 \oplus_2 v_2, \ldots,$ $r_n := r_{n-1} \oplus_n v_n$, such that r_n stores the value of the whole expression. Thus, Jimple instructions generally only take *variables* (not expressions) as arguments.

A *Jimple* program is a collection of methods, with at least one `main` method that corresponds to the program's entry point. A *Method* has a name, a typed signature, and a body consisting of a sequence of instructions, which include: *i*) *Control flow* instructions: unconditional (**`goto`**) and conditional (**`if`**, **`switch`**) jumps to a location with a given label within the method; **`return`** to the caller. *ii*) *Assignment* instructions perform any kind of expression evaluation using the SSA form discussed above. *iii*) *Monitor* instructions correspond to the synchronization when entering (**`monitor_enter`**) and exiting a **`synchronized`** block. *iv*) *Call* instructions are also only used in the right-hand side of an assignment. For simplicity, **`invoke`** represents all five variants of call instructions (`static`, `virtual`, etc.) available in bytecode. *v*) The **`noop`** instruction does nothing.

Example 1. Fig. 1a's example in Jimple consists of three methods: `main`, and two anonymous methods for the `Runnable` objects. Each **`synchronized`** block is a pair of matching **`monitor_enter`** and **`monitor_exit`** instructions. The other statements are different variants of **`invoke`**: `invokespecial` for the **`new`** creation expressions, and `invokevirtual` for the `start()` calls. ∎

4.2 The RB Intermediate Language

To streamline the encoding of concurrent behavior into PNs, we introduce the RB intermediate language. In a nutshell, RB (short for Rock Bottom) is a simplified bytecode-like representation that captures flow- and context-sensitive information while abstracting away path-sensitive details.

Syntax of RB. Fig. 2b outlines the syntax of RB. An *RB* program is a collection of procedures; like in Jimple (and Java) we assume that at least one `main` procedure exists. A *Proc* has a unique name, and a body consisting of a sequence of commands marked by **`begin`** and **`end`**. We also assume that each procedure p also defines labels $\underline{\texttt{entry}}_p$ and $\underline{\texttt{exit}}_p$ marking, respectively, p's unique entry and exit points. RB commands include: *i*) *Control flow* commands: unconditional (**`goto`**) and nondeterministic (**`jump`**) jumps; synchronously **`call`** a procedure p; spawn (**`fork`**) and wait for (**`join`**) a parallel thread t running a procedure p. We assume that the thread identifiers in **`fork`** and **`join`** commands are distinct from all other identifiers, and that the `main` procedure runs on thread t_0. *ii*) *Synchronization* commands to **`acquire`** and **`release`** a lock. *iii*) *Access* commands to **`read`** and **`write`** a variable. *iv*) The **`skip`** command does nothing. Given an RB program *RB*, T denotes the set of all *Thread* identifiers, P the set of all *Procedure* identifiers, B the set of all *Label*s (we assume that each command has a unique label), and V the set of all *Variable* names mentioned anywhere in the program.

Semantics of RB. The *state* S of an RB program is a set of tuples (t, p, ℓ, K, R, τ), where $t \in T$ is a thread, $p \in P$ is a procedure, $\ell \in B$ is a command label, $K \subseteq V$ is a set of locked variables, $R \in B^*$ is a sequence of procedure/label pairs, and $\tau \in \{\blacktriangleright, \times\}$ is the thread's termination state. Informally, such a tuple denotes a thread t is ready to run command at label ℓ in procedure p, while holding locks K; τ denotes whether t is still running ($\blacktriangleright$) or has terminated ($\times$); and R is a stack of return locations of pending calls. The *initial state* of an RB program is $\{(t_0, \texttt{main}, \underline{\texttt{entry}}_{\texttt{main}}, \emptyset, \emptyset, \blacktriangleright)\}$.

$$\frac{s = (t, p, \ell\colon c, L, R, \blacktriangleright) \in S \quad c \in \{\texttt{skip}, \texttt{read}\ v, \texttt{write}\ v\}}{S' = S \setminus \{s\} \cup \{(t, p, \ell + 1, L, R, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{call}\ p', L, R, \blacktriangleright) \in S}{S' = S \setminus \{s\} \cup \{(t, p', \underline{\texttt{entry}}_{p'}, L, R + [\ell + 1], \blacktriangleright)\}}$$

$$\frac{s = (t, p, \underline{\texttt{exit}}_p, L, R + [\ell], \blacktriangleright) \in S \qquad \ell \in p'}{S' = S \setminus \{s\} \cup \{(t, p', \ell, L, R, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \underline{\texttt{exit}}_p, L, \emptyset, \blacktriangleright) \in S}{S' = S \setminus \{s\} \cup \{(t, p, \underline{\texttt{exit}}_p, L, \emptyset, \text{✖})\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{goto}\ \ell', L, R, \blacktriangleright) \in S}{S' = S \setminus \{s\} \cup \{(t, p, \ell', L, R, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{jump}\ \ell_1 \ell_2, L, R, \blacktriangleright) \in S \quad \ell' \in \{\ell_1, \ell_2\}}{S' = S \setminus \{s\} \cup \{(t, p, \ell', L, R, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{acquire}\ k, L, R, \blacktriangleright) \in S \qquad k \notin L \quad \forall t', L' \cdot (t', _, _, L', _, _) \in S \wedge t' \neq t \rightarrow k \notin L'}{S' = S \setminus \{s\} \cup \{(t, p, \ell + 1, L \cup \{k\}, R, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{release}\ k, L, R, \blacktriangleright) \in S \qquad k \in L}{S' = S \setminus \{s\} \cup \{(t, p, \ell + 1, L \setminus \{k\}, R, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{fork}\ p'\ t', L, R, \blacktriangleright) \in S}{S' = S \setminus \{s\} \cup \{(t, p, \ell + 1, L, R, \blacktriangleright)\} \cup \{(t', p', \underline{\texttt{entry}}_{p'}, \emptyset, \emptyset, \blacktriangleright)\}}$$

$$\frac{s = (t, p, \ell\colon \texttt{join}\ t', L, R, \blacktriangleright) \in S \quad s' = (t', p', \ell', L', R', \text{✖}) \in S}{S' = S \setminus \{s, s'\} \cup \{(t, p, \ell + 1, L, R, \blacktriangleright)\}}$$

Fig. 3: Operational semantics of RB. Each rule describes one step of evaluation $S \rightsquigarrow S'$ when executing the command in the rule's premise.

Fig. 3 outlines an operational semantics of RB. Each rule shows how the state S changes after executing a different command. Command `skip` simply continues execution of the thread t to the next location $\ell + 1$. Since RB abstracts away path-sensitive details, `read` and `write` behave exactly like `skip`. However, we still include them as separate commands since they allow us to model, in RB, the different interleavings of read and write operations in a JVM program, and behaviors that depend on them—such as data races. A `call` p' command appends the return label $\ell + 1$ to R and continues executing the callee p' from its entry point. Conversely, when a call terminates, the most recent return label is removed from R and used as next command to execute. If the execution of a procedure p reaches its exit point and R is empty, this makes the whole thread t terminate—denoted by $\tau = \text{✖}$. Both kinds of branching commands, `goto` and `jump`, change the current command label: `goto` does so unconditionally, whereas `jump` nondeterministically picks one of two possible labels. A thread t can execute an `acquire` k only if any other thread t' is *not* holding a lock on k. In contrast, t can execute a `release` k without waiting, as long as it is currently holding a lock on k. Finally, a `fork` p' t' adds a new tuple $(t', p', \underline{\texttt{entry}}_{p'}, \emptyset, \emptyset, \blacktriangleright)$ to the state S, corresponding to a new thread t' starting to execute procedure p'. And a `join` t' can execute only when thread t' has terminated.

Example 2. When Fig. 1b's example RB program deadlocks, its state S consists of the tuples: $(\texttt{t_12}, \texttt{r_12}, \ell_7, \{\texttt{lock1}\}, [], \blacktriangleright)$, $(\texttt{t_21}, \texttt{r_21}, \ell_{11}, \{\texttt{lock2}\}, [], \blacktriangleright)$, $(t_0, \texttt{main}, \underline{\texttt{exit}}_{\texttt{main}}, \emptyset, [], \text{✖})$. Neither ℓ_7 nor ℓ_{11} can execute, because each thread state invalidates the other's progress precondition. ∎

4.3 Petri Nets

A Petri net is a tuple (Π, Δ, A, I), where Π is a set of *places*, Δ is a set of *transitions*, $A \subseteq (\Pi \times \Delta) \cup (\Delta \times \Pi)$ is a set of *arcs* connecting transitions to places and places to transitions, and $I\colon \Pi \rightarrow \mathbb{N}$ is an *initial marking*. The *preset* $pre(\delta)$ of a transition $\delta \in \Delta$

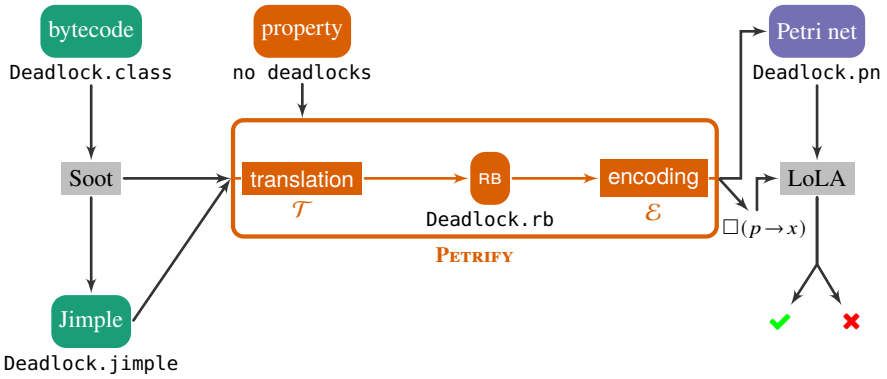


Fig. 4: An overview of how PETRIFY works.

is the set of places π such that $(\pi, \delta) \in A$; and the *postset* $post(\delta)$ is the set of places π such that $(\delta, \pi) \in A$. It is customary to picture a PN as a graph where places are circles, transitions are boxes, and arcs are arrows. Fig. 1 displays the running example's PN using this notation; the black disks denote the net's initial marking.

Semantics of Petri nets. The *state* of a PN is a *marking* $m \colon \Pi \to \mathbb{N}$, which denotes how many *tokens* $m(\pi) \geq 0$ each place $\pi \in \Pi$ holds. A transition $\delta \in \Delta$ is *enabled* in a state m whenever $m(\pi) > 0$ for every place π in $pre(\delta)$; in other words, all places that connect to t are marked with at least one token. Whenever a transition δ is enabled, it can nondeterministically *fire*. If δ fires when the net is in state m, the new marking m' is such that $m'(p) = m(p) - 1$ for every place $p \in pre(\delta)$, and $m'(p) = m(p) + 1$ for every place $p \in post(\delta)$. In other words, each place in δ's preset consumes one token, whereas each place in δ's postset acquires one additional token. If no transition is enabled in a marking m, the PN is in a *deadlock*.

According to these definitions, a PN determines a set of possible *firing sequences*, corresponding to all sequences of transitions that can be triggered from the initial marking; each firing sequence corresponds to a sequence of markings $m_0 \vdash m_1 \vdash \ldots$ that begins with $m_0 = I$. The set of all firing/marking sequences denotes a PN's *semantics*.

Example 3. From its initial marking, Fig. 1c's PN may reach the state where places $(\texttt{t_12}, \ell_7)$, $(\texttt{t_21}, \ell_{11})$, $(\texttt{t_12}, \texttt{lock1})$, $(\texttt{t_21}, \texttt{lock2})$ are marked. From that state, no transition is possible: place $(\texttt{t_12}, \texttt{lock2})$ is unmarked, which disables transition $(\texttt{t_12}, \ell_7)$; and place $(\texttt{t_21}, \texttt{lock1})$ is unmarked, which disables transition $(\texttt{t_21}, \ell_{11})$. ∎

5 How PETRIFY Works

This section details PETRIFY's approach. As outlined in Fig. 4, PETRIFY inputs a program J in bytecode; precisely, it targets the Jimple format offered by the Soot static analyzer. PETRIFY translates J into a corresponding RB program $R = \mathcal{T}(J)$ (Sec. 5.1); it then encodes R's semantics as a Petri net $P = \mathcal{E}(R)$ (Sec. 5.2); it also produces suitable temporal logic properties that can be passed as input, together with P, to a PN analyzer (e.g., LoLA) to detect concurrency issues (Sec. 5.3).

5.1 Translation of Jimple into RB

Given a bytecode program (in Jimple format) J, PETRIFY builds an RB program $\mathcal{T}(J) = R$ that captures a flow- and context-sensitive, path-insensitive approximation of J's behavior.

Modeling Aliasing. Consider any identifier n that appears in J, including variable identifiers and method names. Let $\eta(n)$ denote n expressed in a form that is unambiguous throughout the program: for example, if `o.m` denotes a method `m` of an object `o` of class `0`, η(`o.m`) denotes the fully qualified name `0.m` with a suffix that distinguishes it from other overloaded variants. Let $\varphi \triangleq \bigcup_{\iota:\, Var}\{\eta(\iota)\}$ denote the set of unique variable identifiers anywhere in J—which we'll call "vars" for short. And let $\mathcal{L}$ denote the set of all locations in J. PETRIFY's translation of bytecode uses a *may point to* analysis as follows. Let $\iota \in \varphi$ be a var used at location $\ell \in \mathcal{L}$ in J; $\mu_{\iota,\ell} \subseteq \varphi$ denotes the *set* of vars that may be aliased to ι according to the may point to analysis. The may-alias sets $\mu_{\iota,\ell}$ determine an equivalence relation $\simeq^+ \subseteq (\varphi \times \mathcal{L}) \times (\varphi \times \mathcal{L})$ among vars as follows: first, $\simeq \subseteq (\varphi \times \mathcal{L}) \times (\varphi \times \mathcal{L})$ is the reflexive and symmetric relation defined by $\iota_1, \ell_1 \simeq \iota_2, \ell_2$ iff $\mu_{\iota_1,\ell_1} \cap \mu_{\iota_2,\ell_2} \neq \emptyset$; then, $\simeq^+$ is the transitive closure of $\simeq$: $\iota, \ell \simeq^+ \iota', \ell'$ iff there exist $\iota_1, \ell_1, \ldots, \iota_n, \ell_n$ such that $\iota_1 = \iota$, $\ell_1 = \ell$, $\iota_n = \iota'$, $\ell_n = \ell'$, and $\iota_k, \ell_k \simeq \iota_{k+1}, \ell_{k+1}$ for all $1 \leq k < n$. Finally, let $\alpha(\iota, \ell)$ be a unique identifier that corresponds to the equivalence class of φ according to $\simeq^+$ that ι, ℓ belongs to. Intuitively, α assigns the same identifiers to any two vars in J iff they may be aliased.

RB Encoding. Tab. 1 outlines how PETRIFY translates Jimple statements into RB commands. Methods become procedures in RB, whose bodies get translated statement by statement. Conditional jumps (**`if`** and **`switch`**) become nondeterministic jumps (**`jump`**) in RB. This makes all control-flow paths in the Jimple program feasible in the RB program. Lock acquisition statements (**`monitor_enter`** and **`monitor_exit`**) become the corresponding commands in RB (**`acquire`** and **`release`**). Finally, an assignment translates to a translation of its right-hand side expression, followed by a **`write`** to its target. The translation of non-call expressions is straightforward, as it corresponds to a read of the variables involved in the expression. An **`invoke`** translates to **`read`**s of its actual arguments and possibly its target, followed by a **`call`** to the RB procedure $\eta(o.m)$ translating the called method $o.m$. An exception is for calls of methods `start` and `join` on targets of type `Thread`: these become **`fork`** and **`join`** respectively.

Soundness and Precision of the RB Encoding. The translation $R = \mathcal{T}(J)$ of J is *sound* if the feasible execution paths in R are a *superset* of those in J; conversely, an unsound translation may omit error paths that are possible in the original program. The translation is *precise* if the feasible execution paths in R are a *subset* of those in J; conversely, an imprecise translation may include spurious error paths that are impossible in the original program. Let's summarize how the translation scheme affects soundness and precision. *i*) Translating conditionals with nondeterministic **`jump`**s is sound but path-insensitive; hence, it generally involves a loss of *precision*. *ii*) The translation of lock operations in J with **`acquire`** and **`release`** in R is sound provided the lock is not used *reentrantly*. According to Fig. 3, RB does not allow a thread to acquire a lock on k if it already holds a lock on it; thus, R may omit such executions even if they are possible in J.[1] *iii*) PETRIFY's

[1] Soundly modeling reentrant locks would require a stack-like counting mechanism, which goes beyond the expressiveness of plain PNs; hence, it belongs to future work.

JIMPLE s	RB TRANSLATION $\mathcal{T}(s)$
method $m : t_1 \ldots t_n \to t \; \{B\}$	**def** $\eta(m(t_1 \ldots t_n))$ **begin** $\mathcal{T}(B)$ **end**
$s_1; s_2$	$\mathcal{T}(s_1); \mathcal{T}(s_2)$
noop	**skip**
goto ℓ'	**goto** $\bar{\ell}'$
if v ℓ'	**read** $\alpha(v)$; **jump** $\curvearrowright$ $\bar{\ell}'$
switch v $(v_1 : \ell_1) \ldots (v_n : \ell_n)$	**read** $\alpha(v)$; **read** $\alpha(v_1)$; **jump** $\curvearrowright \bar{\ell}_1; \ldots;$ **read** $\alpha(v_n)$; **jump** $\curvearrowright \bar{\ell}_n$
$v := e$	$\mathcal{T}(e)$; **write** v
return v	**read** $\alpha(v)$; **goto** *exit*
return	**goto** *exit*
monitor_enter v	**acquire** $\alpha(v)$
monitor_exit v	**release** $\alpha(v)$
c : *Const*	–
$v_1 \oplus v_2$	**read** $\alpha(v_1)$; **read** $\alpha(v_1)$
$\odot\, v$	**read** $\alpha(v)$
invoke m o $a_1 \ldots a_n$	**read** $\alpha(v_1); \ldots;$ **read** $\alpha(v_n)$; **read** $\alpha(o)$; **call** $\eta(o.m)$
invoke `start` t : `Thread`	**fork** $\eta(t.\texttt{start})$ $\alpha(t)$
invoke `join` t : `Thread`	**join** $\alpha(t)$

Table 1: Translation $\mathcal{T}$ of Jimple instructions (top) and expressions (bottom) into RB commands. $\bar{\ell}$ denotes the location in the RB program that translates the instruction at location ℓ in the Jimple program; $\curvearrowright$ denotes the location of the next command; $\eta(n)$ denotes a unique form of identifier n; and $\alpha(v)$ denotes a group of variables that may be aliased (see the text for a precise definition).

sound translation of call instructions depends on whether Soot can retrieve the body of the invoked closure object in `invokedynamic` instructions. This is possible in simple cases such as Fig. 1's example, where `invokedynamic` is used to execute the `Runnable` lambdas `r_12` and `r_21`; more complex instances of `invokedynamic` would become **skip** in R, which introduces unsoundness in general. *iv*) Other bytecode instructions that are not listed in Fig. 2a are currently unsupported by PETRIFY. The translation replaces any unsupported instruction I with a **skip**, which means that R doesn't model I's semantics. This may result in a loss of soundness or precision, depending on what execution paths the unsupported instruction does enable or block. *v*) Two variables v_1, v_2 that may be *aliased* in J are lumped together into a single variable $v = \alpha(v_1) = \alpha(v_2)$ in R; this may introduce a loss of *soundness* or, more commonly, *precision*. In practice, these limitations mainly imply the lack of support for certain program features. As our experiments in Sec. 6 show, PETRIFY remains applicable on broad range of Java programs following different concurrency patterns and features. While we plan to remove some limitations in future work, PETRIFY's capabilities are consistent with the pragmatic approach of making program analysis work with realistic programs despite theoretical limitations [23].

Example 4. Fig. 1b shows PETRIFY's RB translation of Fig. 1a; for simplicity, Fig. 1b omits the **write**s of variables `r_12`, `r_21` (assignments on lines 4, 5) and the **read**s of variables `r_12`, `r_21` (**new** expressions on lines 6, 7). Since the program does not have data-dependent path conditions, and there is no aliasing among variables, the translation to RB is sound *and* precise. The RB program consists of three procedures: `main`, and two `Runnable` anonymous functions that are each thread's `run()` method. ∎

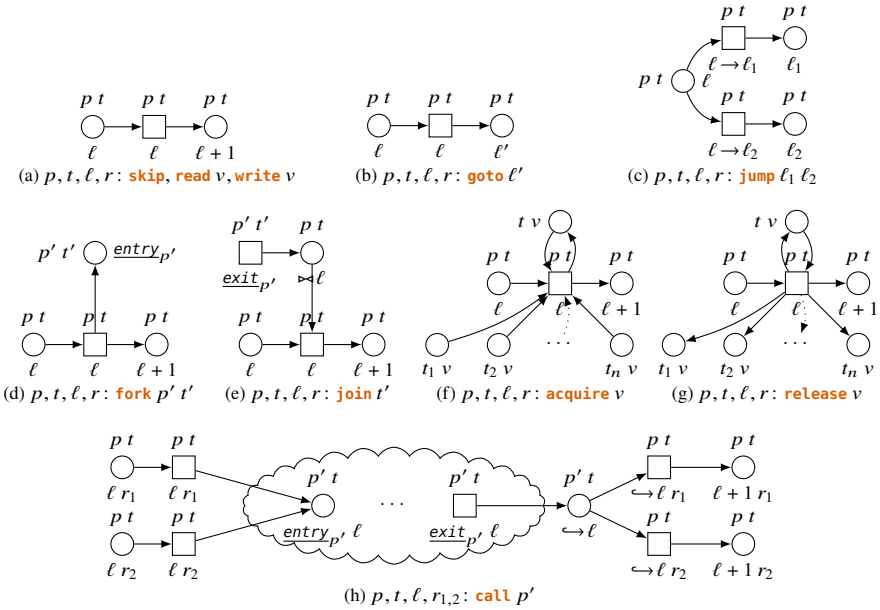


Fig. 5: Fragments of Petri nets encoding RB commands.

5.2 Encoding of RB into Petri Nets

Given an RB program R with threads T, procedures P, labels B, and variables V, PETRIFY builds a PN $N = \mathcal{E}(R) = (\Pi, \Delta, A, I)$ that over-approximates R's semantics. The general idea is that when the command at location ℓ in procedure p is ready to be executed by thread t, the corresponding place (p, t, ℓ, r) is marked. The component r denotes the *caller* location of p, which is $\top$ if p is being executed on the main thread t_0 or on a freshly forked thread. Then, the corresponding transition (p, t, ℓ, r) fires when the command executes; the transition's postset denotes the commands that will be able to execute after c. With this approach, each combination of procedure, thread, and call site in R corresponds to a set of connected nodes (places and transitions) in PN N.

Formally, the set Π of *places* of PN N includes, for every combination of location $\ell \in B$, procedure $p \in P$, thread $t \in T$, and return label $r \in B \cup \{\top\}$: *i*) A place (p, t, ℓ, r) if ℓ is a location in p and one of the following holds: *a*) p is `main`, t is t_0, and r is $\top$; or *b*) there is a **`fork`** p t somewhere in the program, and r is $\top$; or *c*) there is a **`call`** p at location $r \neq \top$ in the program. *ii*) A return place $(p, t, \hookrightarrow \ell)$ if ℓ is the location of a **`call`** p command. *iii*) A join place $(p, t, \bowtie \ell, r)$ if ℓ is the location of a **`join`** t' command. *iv*) A place (t, v) for every variable $v \in V$ such that there is an **`acquire`** v or a **`release`** v somewhere in the program. The set Δ of *transitions* includes, for every combination of location $\ell \in B$, procedure $p \in P$, thread $t \in T$, and return label $r \in B \cup \{\top\}$: *i*) A transition (p, t, ℓ, r) for every place $(p, t, \ell, r) \in \Pi$. *ii*) A return transition $(p, t, \hookrightarrow \ell, r)$ if ℓ is the location of a **`call`** p command; *iii*) Two jump transitions $(p, t, \ell \to \ell_1, r)$ and $(p, t, \ell \to \ell_2, r)$ if ℓ is the location of a **`jump`** ℓ_1 ℓ_2 command. The *initial marking* I has one token in the place $(\texttt{main}, t_0, \underline{\texttt{entry}}_{\texttt{main}}, \top)$—the program's unique entry point—and one token in each place (t, v)—denoting that all locks are initially not held by any thread.

For each command c at location ℓ in procedure p executed by thread t with return location r, Fig. 5 shows the set A of *arcs* that capture the command's semantics. For readability, Fig. 5 omits the caller location r for commands where it's the same in all nodes (i.e., all commands but calls). For brevity, we only describe the most complex fragments: *a*) Command `fork` p' t' puts one token into the next location's place, and one token into the forked thread's entry point's place $(p', t', \underline{\texttt{entry}}_{p'}, r)$. This way, the forked thread's computation can proceed in parallel to the forking thread's. *b*) Conversely, `join` t' can fire only when the exit transition $(p', t', \underline{\texttt{exit}}_{p'}, r)$ of thread t' fires (where p' is the procedure that thread t' is running). *c*) Command `acquire` v's transition can fire only if *all* places (t', v) are marked, denoting that no thread t' holds a lock on v. When it fires, it puts back a token only in place (t, v) to indicate that t holds a lock on v. *d*) Command `release` v's transition can fire only if place (t, v) is marked, denoting that thread t holds a lock on v. When it fires, it puts back a token in all places (t', v), thus allowing other threads to acquire a lock on v. *e*) When the transition (p, t, ℓ, r) of command `call` p' fires, it puts a token in the callee's entry place $(p', t, \underline{\texttt{entry}}_{p'}, \ell)$. Then, execution continues in the subnet corresponding procedure p' called in thread t at call site ℓ. Then, when transition $(p', t, \underline{\texttt{exit}}_{p'}, \ell)$ fires, signaling that the subnet's execution has terminated, a token first goes into return place $(p', t, \hookrightarrow \ell)$; then one of the transitions $(p, t, \hookrightarrow \ell, r)$, for all possible callers of p', nondeterministically fires. Fig. 5h pictures two callers $r_1 \neq r_2$, where either of the return transitions $(p, t, \hookrightarrow \ell, r_k)$, $k = 1, 2$, may fire when the subnet terminates execution. This nondeterministic encoding is the only aspect of Petrify's encoding of RB programs that loses precision: a PN cannot store an unbounded stack of return locations,[2] and hence Petrify overapproximates it.

Example 5. Fig. 1c shows the PN encoding of Fig. 1b and its initial marking. For readability, `m` abbreviates `main`, and we omit the caller in nodes since it's $\top$ everywhere. The PN consists of three subnets, corresponding to the `main` (middle), `r_12` (top), and `r_21` (bottom) procedures; as well as four additional places (`t_`xy, `lock`m) that denote when `t_`xy is holding a lock on `lock`m. The arcs connecting `main` to the other subnets mark the spawning of each thread in the main thread t_0; and the colored arcs connecting transitions in the `r_12` and `r_21` subnets to the places (`t_`xy, `lock`m) with matching color synchronize the corresponding lock commands. ∎

Lemma 1 (Correctness of Petri net encoding). *Let P be an RB program and $N = \mathcal{E}(P)$ its PN encoding defined by Petrify. Then, each sequence of state transitions according to P's operational semantics corresponds to a marking sequences according to N's semantics. Therefore, the semantics of N is a sound over-approximation of P's semantics.*

5.3 Encoding Properties and Implementation

The trace semantics of the PN $N = \mathcal{E}(\mathcal{T}(J))$ built by Petrify is a path-insensitive approximation of the trace semantics of the input bytecode program J. Thus, N can be analyzed with any standard PN analyzer, such as LoLA [31]. While this approach gives flexibility, since one may formalize a wide range of properties as formulas in the temporal logic supported by the analyzers, doing so requires some knowledge of how

[2] PNs with inhibitor arcs would be able to simulate this without loss of precision; however, their reachability problem becomes undecidable [9].

PETRIFY encodes programs into PNs. To improve PETRIFY's usability, we built in support for common concurrency bugs: *deadlock*, *livelock*, and *non-termination*.

We implemented the PETRIFY technique in a command-line tool called jPETRIFY. jPETRIFY inputs a bytecode program J, a property to be verified (a temporal logic formula, or absence of deadlocks, livelocks, or termination), uses Soot to analyze Jimple code and its control-flow, and produces a PN $P = \mathcal{E}(\mathcal{T}(J))$ as described in previous sections.

6 Experimental Evaluation

Evaluation goal. The goal of this experimental evaluation is to assess PETRIFY's practical feasibility, highlighting its capabilities, limitations, and complementarity in comparison to other automated Java verification tools that can analyze similar concurrency properties.

6.1 Subjects and Setup

Comparable tools. According to Sec. 3's discussion, we focus on JPF (Java Pathfinder [10]) and JaDA [19] for the following reasons: *i*) they support checking (absence of) deadlocks out of the box—the same properties currently supported by jPETRIFY; *ii*) their public repositories (especially JaDA's) include numerous examples of Java programs that demonstrate their capabilities in deadlock detection; *iii*) they represent two distinct approaches: JPF is based on model-checking, on top of which it implements an array of analysis techniques, whereas JaDA uses type-based static analysis and is specialized on deadlock detection; *iv*) their implementations have different levels of maturity: JPF is a mature tool with over 20 years of history, whereas JaDA is more recent and less polished, but with a ready-made, usable implementation. Our experiments are not meant as a direct *comparison* between jPETRIFY and these two tools; however, they will show that jPETRIFY works successfully on several examples that were designed for JPF and JaDA, indicating that our approach is viable and has potential, as well as its limitations compared to the state of the art.

Subjects. We evaluated jPETRIFY's capabilities on 39 example concurrent programs with shared locks. The leftmost part of Tab. 2 lists these programs, which belong to 7 groups: *i*) 2 are from *JPF*'s repository; *ii*) 14 are from *JaDA*'s repository [12]; *iii*) 2 are from the JaConTeBe [22] *benchmarks*; *iv*) 7 are *variants* of other examples: 2 are variants of programs in group *JaDA* (`ClassicDeadlockN` is a generalization of `ClassicDeadlock`, and `PhilTable` is a dining philosophers variant), and the other 5 use recent Java language features (up to Java 21); *v*) 11 are examples that specifically scale up the *size* and complexity of the verification problems by introducing nested loops, several threads, sequences of blocks, and unbounded recursion; *vi*) 3 are adaptations of other classic examples translated to *Kotlin*. We selected deadlock examples from JPF's and JaDA's repositories that don't use features unsupported by jPETRIFY (especially arrays, which result in a major loss of precision); for the same reason, we refactored some of the JaDA examples to make them compatible with jPETRIFY without changing their behavior. As indicated in Tab. 2, 3 of the subjects are correct, and 36 include a concurrency (deadlock) bug. Therefore, the evaluation tests the tools' capabilities both to precisely find bugs and to soundly verify correctness.

Challenges. Although most benchmark programs are relatively small in terms of lines, methods, or classes, they are far from trivial, and introduce various kinds of challenges to even state-of-the-art tools. In particular: *i*) Group *variants* uses language features

(sealed classes, records, etc.) that have only been introduced in recent versions of Java; for approaches that work on the source level (like JPF and JaDA), supporting new language features is a major challenge that often goes beyond "mere" engineering. Notably, program `VirtualThreads` specifically targets a new *concurrency* feature of Java that jPetrify supports out of the box. On the other hand, program `PhilTable` uses a lot of aliased references, which challenge the tools' abstraction of this feature. *ii*) Programs in group *size* feature different forms of complex control flow, which challenges the *scalability* of concurrency analysis: unbounded recursive synchronous calls (`RecUnbounded`, `Chordv2`); recursive thread spawning and joining (`RecJoin`); thread synchronization with unbounded loops (`WhileCnt2N2T`), deeply nested calls (`Interleaving`), and nested locking of many lock variables (`Nested200T2`); and an 8-thread generalization of the dining philosophers problem (`DiningP8T`). The remaining programs in this group feature behaviors that challenge concurrency analysis with complex *dependency* patterns (`InnerThreads`, `JoinUnderLock`, `RecursiveJoinUnderLockDeadlock`), such as circular waiting between parent and child threads, or require a context-sensitive analysis (`DeeperContextDeadlock`).

Setup. We ran each example with jPetrify, JPF, and JaDA with 15-minute timeout; we classify a tool's output as: *i*) success ✔ if it correctly detects a deadlock or establishes that there is none; *ii*) false negative ━ if it misses an existing deadlock; *iii*) false positive ✚ if it reports a spurious bug; *iv*) time out/out of memory ⌛; *v*) failure due to unsupported features ⊘. All experiments ran on an Apple Macbook M3 Max with 36 GB RAM and macOS 15.6.1. jPetrify used Soot 4.6.0 and LoLA 2.0; JPF commit `0f2f2` used Java 11; the JaDA tool was run through its website[3] since it is not available for download.

6.2 Results

Performance. Tab. 2 shows that jPetrify's encoding is succinct, with 2.4 (6624/2738) places per source line of code on average. Also as a result of this succinctness, verification is fast in most examples, typically taking under a second. In contrast, generating the PN encoding from bytecode takes significantly more time; however, most of the generation time (3.8 *minutes* per subject) is taken by Soot to extract from bytecode the information needed for the rb encoding, whereas jPetrify's actual encoding is much faster (4.2 seconds per subject). The overall end-to-end performance of jPetrify is acceptable, but there are clear margins for improvement, in particular as we invest more time into optimizing its bottleneck interaction with Soot. It is also encouraging that the two outliers (`Nested200T2` and especially `PhilTable`) that took considerably more time resulted in larger-than-average PN models that similarly challenged encoding and verification; in other words, they are intrinsically more complex examples. We did not collect the running time of JPF or JaDA: a proper comparison of performance would require controlled experimental conditions (not possible for JaDA whose implementation is not open source), and would be of limited interest for a prototype tool like jPetrify.

jPetrify correctly verified (detecting a deadlock or confirming that none exists) 38 examples. The lone exception is `PhilTable`, where Soot's alias analysis induces an unnecessarily large number of lock-thread combinations, which results in a huge state space on which jPetrify runs out of memory. This example showcases a current limitation

[3] `http://jada.cs.unibo.it/`

GROUP	NAME	EXP	LOC	CLASSES	METHODS	|PN|	SOOT [s]	ENC [s]	VER [s]	OUTCOME JPE	JPF	JADA
JPF	BankTransfer	🐞	18	2	4	91	175.9	0.5	0.1	✔	✔	⊘
	DiningPhil	🐞	40	3	3	133	212.1	1.5	0.5	✔	✔	−
JaDA	AnythingTest	🐞	36	2	5	77	172.4	0.4	0.2	✔	✔	✔
	BuildNetwork	🐞	52	3	4	112	157.1	0.5	0.1	✔	✔	−
	Chord	🐞	37	1	7	67	306.8	2.5	0.9	✔	✔	✔
	ClassicDeadlock	🐞	48	1	6	73	148.4	0.4	0.1	✔	✔	✔
	ClassicPhil	🐞	51	2	8	217	148.6	0.6	0.2	✔	⧖	✔
	DanglingThreads	🐞	59	1	10	91	150.4	0.5	0.1	✔	✔	✔
	DeadlockTwo	🐞	34	1	5	112	149.8	0.5	0.1	✔	✔	−
	GuardedLocks	✅	30	1	3	64	165.3	0.5	0.1	✔	✔	+
	MayNotHappenInParallel	✅	26	1	3	53	166.1	0.4	0.2	✔	✔	✔
	NetworkAllP	🐞	52	3	4	112	147.8	0.4	0.1	✔	✔	−
	PhilTableP	🐞	60	4	5	203	331.1	3.7	0.8	✔	⧖	−
	SimpleWhile	🐞	31	2	3	80	155.0	0.4	0.1	✔	⧖	✔
	StaticFields	🐞	28	1	3	65	167.1	0.5	0.1	✔	✔	−
	SynchMethod	🐞	21	1	3	46	169.3	0.5	0.1	✔	✔	✔
benchmarks	Deadlock1	🐞	40	4	3	72	148.7	0.4	0.1	✔	✔	−
	Deadlock2	🐞	25	2	3	51	148.5	0.3	0.1	✔	✔	✔
variants	ClassicDeadlockN	✅	42	1	5	73	322.4	2.9	0.8	✔	✔	✔
	PhilTable	🐞	59	2	9	234	162.0	88.9	214.5	⧖	⧖	✔
	Records	🐞	55	4	7	71	314.9	3.2	0.9	✔	⊘	⊘
	Sealedclasses	🐞	57	6	7	71	313.4	2.9	0.7	✔	⊘	⊘
	Switch	🐞	59	4	7	91	314.5	4.1	1.0	✔	⊘	⊘
	VirtualThreads	🐞	57	3	5	12	157.9	0.68	0.2	✔	⊘	⊘
	Yield	🐞	50	3	6	73	317.0	2.8	0.7	✔	⊘	⊘
size	Chordv2	🐞	37	1	7	47	306.3	2.8	0.7	✔	⧖	−
	DeeperContextDeadlock	🐞	65	1	8	186	303.8	4.1	0.7	✔	✔	−
	DiningPhilP8T	🐞	60	3	19	666	276.6	4.9	0.9	✔	⧖	−
	InnerThreads	🐞	32	1	4	54	308.8	2.4	0.5	✔	✔	✔
	Interleaving	🐞	155	1	30	547	169.1	1.1	0.3	✔	⧖	−
	JoinUnderLock	🐞	22	1	3	40	307.7	2.4	0.7	✔	✔	−
	Nested200T2	🐞	1015	1	6	1870	271.0	11.8	4.1	✔	✔	⧖
	RecJoin	🐞	44	1	6	175	284.9	2.8	0.7	✔	✔	⧖
	RecUnbounded	🐞	69	3	6	274	297.6	3.4	0.7	✔	✔	−
	RecursiveJoinUnderLock	🐞	35	1	5	89	298.4	2.9	0.7	✔	✔	−
	WhileCnt2N2T	🐞	34	1	5	91	282.0	3.1	0.6	✔	⧖	−
Kotlin	Chord	🐞	41	1	6	88	148.3	0.5	0.1	✔	✔	⊘
	ClassicDeadlock	🐞	29	1	2	64	147.9	0.5	0.1	✔	✔	⊘
	When	🐞	45	1	4	89	148.4	0.5	0.1	✔	✔	⊘
total			2738	76	238	6624	8818.8	164.2	234.2			
average			70	2	6	170	226.1	4.2	6.0			

Table 2: Experimental subjects and results. For each GROUP of programs used to evaluate jPETRIFY, the table lists the NAME of each program in the group, whether the program is correct ✅ has a deadlock 🐞, its size in lines of code LOC, number of CLASSES and METHODS. The table also reports the size |PN| in number of places of jPETRIFY's PN model of the program, the times (in seconds) for SOOT's analysis, jPETRIFY's PN ENCoding, and LoLA's VERification. The right-hand side details the OUTCOME of verification with each tool: jPETRIFY, JPF, and JaDA: success ✔, false negative −, false positive +, time out/out of memory ⧖, and failure due to unsupported features ⊘.

of jPETRIFY, which depends on the precision of Soot's alias analysis. The flip side is that plugging in a bespoke alias analysis—something for future work—in jPETRIFY's implementation will improve its precision without changes to the underlying technique.

Language feature support. Since jPETRIFY targets bytecode, it can verify programs in group *variants* that use features that have only been available in recent Java versions. In contrast, both JPF and JaDA fail to analyze 5 examples in *variants* since they do not support recent language features; for the same reason, JaDA also fails on the `BankTransfer` example that uses lambdas. jPETRIFY can also analyze the 3 examples written in *Kotlin*; somewhat surprisingly, JPF can also analyze them, but only because they use basic features that compile to a small bytecode instruction set that JPF can input. Instead, JaDA is strictly

limited to programs written in Java. As explained above, we deliberately selected these examples to demonstrate jPetrify's adaptability to recent Java features and other JVM languages; jPetrify has other language limitations, which we discussed in Sec. 5.

Scalability. jPetrify successfully analyzed 4 examples in group *size* and 3 examples in group *JaDA* on which JPF ran out of memory. This indicate that JPF struggles to scale to programs with significant usage of recursion (`ClassicPhil`, `DiningPhilP8T`, `Chordv2`, `PhilTableP`) or a large number of interleavings (`SimpleWhile`, `WhileCnt2N2T`, `Interleaving`). Interestingly, JPF times out too on the one example `PhilTable` where jPetrify runs out of memory. While JaDA's abstractions make it a generally more scalable tool, it still timed out on 2 examples in group *size* that jPetrify can analyze. In all, these examples indicate that the kinds of complex behaviors that jPetrify can analyze are somewhat complementary to the focus of tools based on different approaches.

Soundness and precision. In all our experiments neither jPetrify nor JPF produced any false positives or false negatives. While Petrify relies on approximations, which we presented formally in the paper, that may break soundness or precision, these experiments indicate that jPetrify remains practically applicable on numerous examples despite its theoretical limitations. In contrast, JaDA incurred 15 false negatives and 1 false positive in the experiments. The false negatives indicate that its abstractions are not sound in general, since they sometimes omit program interactions that are feasible and may trigger a deadlock. The false positive (`GuardedLocks`) is simply a result of an imprecise overapproximation of the possible sequential executions under locking.

Evaluation summary. Our evaluation indicates that jPetrify is applicable to programs using modern Java features, including concurrency features such as virtual threads. Despite being a prototype, jPetrify can tackle programs with complex control flow (nested conditionals, loops, recursion) and dynamic, unbounded thread creation. Its main limitations are the lack of support for Java features like exceptions and arrays, and its imprecision in the presence of complex aliasing. Overall, its current capabilities and limitations complement other Java concurrency analysis tools.

Limitations. Our experiments are are not meant to be a direct comparison between jPetrify and JPF and JaDA, both because JaDA and, especially, JPF are much more mature tools, and because their characteristics are largely complementary.

7 Conclusions

This paper presented Petrify: a technique to analyze concurrency properties of Java programs based on model-checking an encoding of a flow- and context-sensitive, path-insensitive approximation of the bytecode semantics. Two characteristics distinguish Petrify's approach from other automated verification techniques for concurrent Java: *i*) Petrify uses Petri nets (PNs) to approximate the program's concurrent behavior and to apply model checking. *ii*) Petrify operates on Java bytecode rather than source code. This brings several practical advantages, in particular in terms of robustness and flexibility. The experiments we discussed in the paper demonstrate the practical advantages brought by Petrify's design choice, as well as its current limitations. Even though jPetrify is still a prototype tool, we could find several examples of programs that challenge other more mature Java automated concurrency verification tools that it can analyze successfully.

References

1. Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: RacerD: compositional static race detection. Proc. ACM Program. Lang. **2**(OOPSLA), 144:1–144:28 (2018)
2. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: Integrated Formal Methods – 13th International Conference (IFM) 2017, Turin, Italy, September 20–22, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2017)
3. Bourdil, P.A., Berthomieu, B., Dal Zilio, S., Vernadat, F.: Symmetry reduction for time Petri net state classes. Science of Computer Programming **132** (09 2016). https://doi.org/10.1016/j.scico.2016.08.008
4. de Carvalho Gomes, P., Gurov, D., Huisman, M., Artho, C.: Specification and verification of synchronization with condition variables. Sci. Comput. Program. **163**, 174–189 (2018)
5. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying java bytecode. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 183–190. Springer (2018)
6. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019)
7. Farzan, A., Madhusudan, P.: Causal atomicity. In: Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 315–328. Springer (2006)
8. Ferrara, P.: A generic static analyzer for multithreaded Java programs. Softw. Pract. Exp. **43**(6), 663–684 (2013)
9. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling Time in Computing. Monographs in Theoretical Computer Science. An EATCS series, Springer (2012)
10. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000)
11. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011)
12. Java deadlocks examples. https://github.com/abelunibo/Java-Deadlocks (2025)
13. Jorshari, M.H.K., Kokologiannakis, M., Majumdar, R., Nagendra, S.: Optimal concolic dynamic partial order reduction. In: 36th International Conference on Concurrency Theory, CONCUR 2025, Aarhus, Denmark, August 26-29, 2025. LIPIcs, vol. 348, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025)
14. Kahsai, T., Rümmer, P., Schäf, M.: JayHorn: A Java model checker (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 214–218. Springer (2019)
15. Kester, D., Mwebesa, M., Bradbury, J.S.: How good is static analysis at finding concurrency bugs? In: Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 2010, Timisoara, Romania, 12–13 September 2010. pp. 115–124. IEEE Computer Society (2010)
16. Kordon, F., Hulin-Hubard, F., Jezequel, L., Paviot-Adet, E., Nivon, Q., , Amat., N., Berthomieu, B., Dal Zilio, S., , Ding, Z., He, Y., Li, S., Jiang, C., Jensen, P., Srba, J., Thierry-Mieg, Y.: Complete Results for the 2025 Edition of the Model Checking Contest. https://mcc.lip6.fr/2025/results.php (June 2025)

17. Koval, N., Fedorov, A., Sokolova, M., Tsitelov, D., Alistarh, D.: Lincheck: A practical framework for testing concurrent data structures on JVM. In: Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13964, pp. 156–169. Springer (2023)
18. Kulkarni, R., Mathur, U., Pavlogiannis, A.: Dynamic data-race detection through the fine-grained lens. In: 32nd International Conference on Concurrency Theory, CONCUR 2021, Virtual Conference, August 24-27, 2021. LIPIcs, vol. 203, pp. 16:1–16:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
19. Laneve, C., Garcia, A.: Deadlock detection of Java bytecode. In: Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10855, pp. 37–53. Springer (2017)
20. Leavens, G.T.: Tutorial on JML, the Java modeling language. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. p. 573. ACM (2007)
21. Lee, Z.H., Mathur, U.: Efficient decrease-and-conquer linearizability monitoring. Proc. ACM Program. Lang. **9**(OOPSLA2) (Oct 2025). https://doi.org/10.1145/3763123, https://doi.org/10.1145/3763123
22. Lin, Z., Marinov, D., Zhong, H., Chen, Y., Zhao, J.: JaConTeBe: A benchmark suite of real-world Java concurrency bugs (T). In: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015. pp. 178–189. IEEE Computer Society (2015)
23. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015)
24. Ma, X., Wu, S., Pobee, E.B., Mei, X., Zhang, H., Jiang, B., Chan, W.K.: Regiontrack: A trace-based sound and complete checker to debug transactional atomicity violations and non-serializable traces. ACM Trans. Softw. Eng. Methodol. **30**(1), 7:1–7:49 (2021)
25. Mathur, U., Viswanathan, M.: Atomicity checking in linear time using vector clocks. In: ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020. pp. 183–199. ACM (2020)
26. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016)
27. Shenoy, A., Furia, C.A.: jPetrify: implementation and experiments. https://doi.org/10.6084/m9.figshare.30775448 (May 2026)
28. Thokair, M.A., Zhang, M., Mathur, U., Viswanathan, M.: Dynamic race detection with O(1) samples. Proc. ACM Program. Lang. **7**(POPL), 1308–1337 (2023)
29. Tunç, H.C., Mathur, U., Pavlogiannis, A., Viswanathan, M.: Sound dynamic deadlock prediction in linear time. Proc. ACM Program. Lang. **7**(PLDI), 1733–1758 (2023)
30. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986. pp. 332–344. IEEE Computer Society (1986)
31. Wolf, K.: Petri net model checking with LoLA 2. In: Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10877, pp. 351–362. Springer (2018)