SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A generic static analyzer for multithreaded Java programs

P. Ferrara

ETH Zurich, Switzerland

SUMMARY

In this paper we present Checkmate, the first generic static analyzer of multithreaded Java programs based on abstract interpretation. Checkmate can be tuned at different levels of precision and efficiency in order to prove various properties (e.g., absence of divisions by zero and data races), and it is sound for multithreaded programs. It supports all the most relevant features of Java multithreading, such as dynamic thread creation, runtime creation of monitors, and dynamic allocation of memory. The experimental results demonstrate that Checkmate is accurate, and efficient enough to analyze programs with some thousands of statements and a potentially infinite number of threads. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Static Analysis, Generic Analyzers, Abstract Interpretation, Multithreaded Programs

1. INTRODUCTION

Testing and debugging multithreaded programs is particularly difficult [1]. Because of the interleaving of parallel threads during execution, it is hard to expose specific bugs and, even if these are produced, sometimes it is arduous to reproduce them. In addition, some executions may be exposed only by specific compilers, virtual machines, or hardware architectures. On the other hand, concurrency may significantly speed up the execution of programs.

For these reasons, tools that discover bugs in multithreaded software are particularly desirable [2]. They should find classical bugs (e.g., null pointer accesses and divisions by zero) as well as concurrency errors (e.g., data races and deadlocks). Many tools based on static analysis prove a property on all possible executions by over-approximating the semantics of a program. In this context, generic analyzers have been deeply studied. They define the semantics of statements and programs such that the analysis can be plugged with various abstract domains, and can check several properties.

Many generic static analyzers based on abstract interpretation [3, 4] have been formalized and implemented. They can be fitted with different domains, to obtain faster and more approximated or slower and more refined analyses, and in order to analyze different properties. Some examples of these analyzers are `Clousot` [5], `Cibai` [6], and `Julia` [7]. None of these analyzers supports multithreading.

1.1. Contribution

In this paper we present Checkmate*, a generic analyzer of multithreaded Java programs, based on abstract interpretation. It can be plugged with different numerical domains, properties, and memory models. Checkmate combines and implements several approaches [8, 9, 10] into one practical and useful analyzer.

*`http://www.pm.inf.ethz.ch/people/pferrara/Checkmate/`

The main technical features of Checkmate are: (i) it works at bytecode level [11], so it can analyze libraries whose source code is not available, and programs written in other languages that compile to Java bytecode (such as Scala [12]), (ii) it supports the main features of Java multithreading (namely, dynamic unbounded thread creation, runtime creation and management of monitors, and dynamic allocation of shared memory), and (iii) it supports many features of Java, such as strings, arrays, static fields and methods, method-calls in the presence of overloading, overriding and recursion, etc.

The analysis performed by Checkmate is (i) whole-program, i.e., it analyzes a complete program starting from its main method; (ii) context-sensitive, e.g., it tracks information through method calls and conditional branches; (iii) completely automatic, i.e., it does not require any manual annotations, such as contracts [13].

We have applied Checkmate to several case studies and benchmarks to study its precision and efficiency. The experimental results show that Checkmate is (i) precise enough to catch common bugs, (ii) fast enough to be applied to programs containing thousands of statements and a potentially unbounded number of threads.

The paper is structured as follows. In the rest of this section we introduce the running example used throughout this paper. Section 2 discusses related work, while Section 3 briefly sketches some background on abstract interpretation, the happens-before memory model, and its definition in a fixpoint form. Section 4 describes the architecture of Checkmate and discusses the main design choices and limitations. Section 5 reports and discusses the experimental results. Finally, Section 6 concludes and discusses future work.

This paper is based on the work published at the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM '09) [14].

1.2. The Running Example

We introduce a running example to illustrate the main features of Checkmate. The program in Figure 1 runs two threads in parallel. Class System stores an instance of class Account in its public static field a. Account implements a bank account which allows us to withdraw money and obtain its balance. The initial balance is set to 1000, and then MyThread is launched. The two threads are both synchronized on the same monitor. System prints the balance of the account, while MyThread sets the account to null if its balance is less than 100; otherwise it withdraws 100 from the account.

Various properties could be interesting (e.g., the presence of data races and null pointer dereferences), but specific numerical domains and memory models should be applied to successfully analyze these properties. We will show in Section 4 how, using different parameters to Checkmate, we can prove the absence of data races and null pointer dereferences for this running example.

2. RELATED WORK

Static analysis has already been widely and extensively studied. In this section, we briefly sketch its main achievements in the field of multithreading.

2.1. Model Checking

Context-bounded model checking [15, 16] has obtained an important amount of both theoretical and practical results. Starting from the premise that verifying a concurrent program (with a context and synchronization sensitive analysis, when dealing with rendezvous style synchronization primitives) is undecidable [17], a multithreaded program is analyzed until a given context bound, i.e., the number of context switches is limited to n. Instead, Checkmate relies on the idea of abstraction, and it is sound w.r.t. all possible multithreaded executions, allowing for an unbounded number of context switches, all possible multi-core architectures, and also supporting weak memory models.

Thread-modular model checking [18] reduces the explosion of the state space by considering each thread separately. The updates of the shared memory performed by a thread are summarized by a single environment assumption, and this is used by other threads to know the values written in

DOI: 10.1002/spe

```
class System {
  public static Account a=new Account();
  public static int main(String[] args){
    MyThread th=new MyThread();
    System.a.balance=1000;
    th.start();
    synchronized(System.a) {
      System.a.printBalance();
    }
  }
}
class MyThread {
  public void run() {
    synchronized(System.a) {
      int temp=System.a.getBalance();
      if (temp<100)
        System.a=null;
      else System.a.withdraw(100);
    }
  }
}
public class Account {
  public int balance=0;
  public void withdraw(int amount) {
    synchronized(this) {
      this.balance-=amount;
    }
  }
  public int getBalance() {
    return this.balance;
  }
  public void printBalance() {
    System.out.println(this.balance);
  }
}
```

Figure 1. A multithreaded application

parallel. Checkmate also considers each thread separately when computing its semantics, but it can be tuned at different levels of precision and efficiency.

Other authors [19] have proposed intra-procedural analyses which work by summarizing the concurrent behavior of other procedures. They consider only well-synchronized programs, and so they are not sound for all possible multithreaded executions (e.g., programs containing data races).

2.2. Concurrency Properties

Many approaches have focused on particular concurrency properties, in particular on deadlock and data race detection [20]. They usually take into account only sequentially consistent executions, but this is not sound in the presence of weak memory models. In addition, these analyses are specific to a single property.

2.2.1. Data Race Analysis Abadi et al. [21] develop a type system to ensure the absence of data races. This analysis is modular, and it scales, but it requires manual type annotation.

Naik and Aiken [22] apply a must-not alias analysis through a specific type system to check the absence of data races. Race freedom is proved by checking that if two locks must not alias the same monitor, then the accesses to the shared memory must not be on the same location. The overall analysis is not particularly efficient, since it takes more than 3 minutes to analyze only 2 classes.

Kahlon et al. [23] present a model-checking-based analysis to detect data races. The work is composed of three phases: (i) discovering which variables share information, (ii) using a must-alias

DOI: 10.1002/spe

analysis to check the owned monitors when shared variables are accessed, and (iii) reducing the false warnings.

Another data race detector based on model checking is introduced by Henzinger et al. [24]. Its programming language uses atomic sections for synchronization, and it is quite different from the lock-based synchronization of Java. Moreover, the experimental results show that the approach is affected by the state space explosion problem.

2.2.2. Deadlock Detection Many works have focused on the dynamic detection of deadlocks [25, 26, 27]. As usual when testing programs, these tools can find deadlocks during an execution, but they cannot prove their absence for all possible executions of a program.

In the field of static analysis, Williams et al. [28] propose an effective analysis that detects deadlocks on synchronized statements and wait invocations. This analysis makes some assumptions on how a user interacts with libraries. In order to analyze a library it supposes that the client code "well-behaves". In this way, even if a library is validated by this analysis, there may be a deadlock when using it without respecting these assumptions.

Awargal et al. [29] introduce a type system to detect potential deadlocks on synchronized statements at compile time. The information inferred by the static analysis is used to check at runtime only the locks which are not proved to be deadlock-free. The analysis has not been implemented, but only manually validated by studying the speed-up of the runtime that uses this information.

2.3. Other properties

Many static analyses check the absence of null pointer accesses [30, 31, 32, 33], divisions by zero [34, 35], and overflows [36]. Usually these approaches are sound for sequential programs, and they do not support concurrency. On the other hand, they are often more precise than Checkmate, since we do not develop specific analyses for these properties and we apply standard and sometimes rough abstractions. We think that Checkmate could be extended to support these approaches, but this will require the support of new features (e.g., relational numerical domains).

3. BACKGROUND

This section introduces some background which helps with understanding the architecture of Checkmate. In particular, we introduce some basic concepts about abstract interpretation, the happens-before memory model, and its fixpoint computation.

3.1. Abstract Interpretation

Abstract interpretation is a theory to define and soundly approximate the semantics of a program. A concrete semantics, aimed at specifying the runtime properties of interest, is defined. It is then approximated with an abstract semantics that is computable, but still precise enough to capture the property of interest. In particular, the abstract semantics must be composed of an abstract domain, an abstract transfer function, and a widening operator to make the analysis convergent if the abstract domain does not satisfy the ascending chain condition. Abstract interpretation can be applied to develop generic analyzers [37]. In particular, this theory allows one to define a compositional analysis, e.g., a generic analysis that can be instantiated with different numerical domains, and to analyze different properties.

3.2. The Happens-Before Memory Model

Memory models define which behaviors are allowed during the execution of a multithreaded program. In particular, they specify which values written in parallel may be read from the shared memory. The Java memory model [38] is quite complex, especially from the point of view of static analysis. Other memory models have been proposed in the past: Lamport [39] formalized the rule of

DOI: 10.1002/spe

sequential consistency. It is quite simple, but too restrictive in practice. A good compromise is the happens-before memory model [40]; it is an over-approximation of Java's model, and it is simple enough to base a static analysis on it.

The core of the happens-before memory model is a partial ordering between the actions performed by a program. In particular, an action a_1 happens-before another action a_2 if (i) a_1 appears before a_2 in the program order, or (ii) a_2 synchronizes-with a_1 (e.g., a_2 locks a monitor that was released by a_1), or (iii) a_2 can be reached by following happens-before edges starting from a_1. Using this partial order, the happens-before consistency rule is defined. This states that a read of a variable is allowed to see a write on that variable if (i) it is not the case that the read happens-before the write, and (ii) there is no write on the same variable that is executed between the observed write and the read, thereby overwriting the observed value.

3.3. Fixpoint Definition

In [8] we define the happens-before memory model[†] in a way that is amenable to a fixpoint computation, and then we abstract it with a computable semantics. On the concrete domain, the *step* function defines the possible results (following the happens-before consistency rule) of one computational step (atomic at the multithreaded level) of a given thread, for a given multithreaded execution. The *step* function tracks the "synchronizes-with" relation when locking and releasing monitors, and launching new threads as well. Relying on this small step operational semantics, a fixpoint trace semantics $\mathbb{S}^\circ$ computes the semantics of a single thread.

Definition 3.1 (Single-thread semantics $\mathbb{S}^\circ$)
Let σ_0 be the initial state of computation.

$$\mathbb{S}^\circ : [(\Psi \times \Omega \times \mathsf{TId}) \to \wp(\mathsf{St}^{\vec{+}})]$$
$$\mathbb{S}^\circ[\![\mathsf{f}, \mathsf{r}, \mathsf{t}]\!] = lfp^{\subseteq}_{\emptyset} F^\circ$$

where

$$F^\circ : [\wp(\mathsf{St}^{\vec{+}}) \to \wp(\mathsf{St}^{\vec{+}})]$$
$$F^\circ = \lambda \mathsf{T}.\{\sigma_0\} \cup \{\sigma_0 \to \cdots \to \sigma_{\mathsf{i}-1} \to \sigma_{\mathsf{i}} : \sigma_0 \to \cdots \to \sigma_{\mathsf{i}-1} \in \mathsf{T} \wedge \sigma_{\mathsf{i}} \in \mathit{step}(\mathsf{t}, \mathsf{f}, \mathsf{r}, \sigma_{\mathsf{i}-1})\}$$

The intuition behind this formal definition is to compute the semantics of a single thread, given the identifier of the thread (represented by $\mathsf{t} \in \mathsf{TId}$), an execution of parallel threads ($\mathsf{f} \in \Psi$), and some information to track the synchronize-with relation ($\mathsf{r} \in \Omega$). The definition of the trace semantics in a fixpoint form is standard in abstract interpretation.

This single-thread trace semantics is the basis for the definition of the trace semantics $\mathbb{S}^{\|}$ of a multithreaded program.

Definition 3.2 (Multithread semantics $\mathbb{S}^{\|}$)

$$\mathbb{S}^{\|} : [\Psi \times \Omega \to \wp(\Psi \times \Omega)]$$
$$\mathbb{S}^{\|}[\![\mathsf{f}_0, \mathsf{r}_0]\!] = lfp^{\subseteq}_{\emptyset} F^{\|}$$

where

$$F^{\|} : [\wp(\Psi \times \Omega) \to \wp(\Psi \times \Omega)]$$
$$F^{\|} = \lambda\Phi.\{(\mathsf{f}_0, \mathsf{r}_0)\} \cup \{(\mathsf{f}_{\mathsf{i}}, \mathsf{r}_{\mathsf{i}-1}) : \exists(\mathsf{f}_{\mathsf{i}-1}, \mathsf{r}_{\mathsf{i}-1}) \in \Phi : \forall \mathsf{t} \in \mathit{dom}(\mathsf{f}_{\mathsf{i}-1}) :$$
$$\tau \in \mathbb{S}^\circ[\![\mathsf{f}_{\mathsf{i}-1}, \mathsf{r}_{\mathsf{i}-1}, \mathsf{t}]\!], \tau \in \mathsf{St}^{\vec{+}}_{\circ\!\!\rightarrow}, \mathsf{f}_{\mathsf{i}}(\mathsf{t}) = \tau\}$$

This semantics iterates the single thread semantics $\mathbb{S}^\circ$ over all active threads until a fixpoint is reached. Each iteration may, for each thread, expose new values written in parallel, thus causing new executions during the following iteration. Iterating this process until a fixpoint is reached allows us to obtain an approximation of all possible multithreaded executions.

[†]Without considering *out-of-thin-air* values

DOI: 10.1002/spe

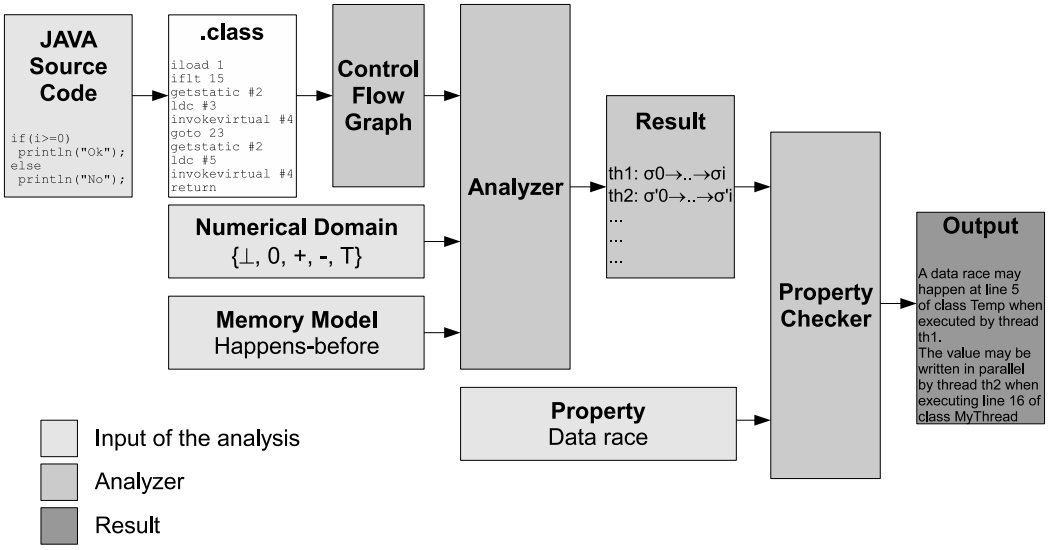


Figure 2. Overall structure of Checkmate

```
public interface MemoryModel {
  public Value get(Reference ref, String field , JVMState state, Statement statement);
  public MemoryModel factory(MultiThreadResult prev, int iteration_number);
}
```

Figure 3. The code of the MemoryModel interface

A similar fixpoint semantics is defined on the abstract domain. The main difference is that it relies on the upper bound operator of the abstract domain to compute one abstract trace that approximates all the concrete executions, introducing approximation while achieving computability. The soundness of this formal system is proved, relying on the abstract interpretation framework. We refer the interested reader to Chapter 3 of [41] for the technical details of these definitions and proofs.

Note that this approach is completely generic w.r.t. the semantics of the programming language, the domains used to track information on numerical values, references, etc., and the property of interest. This will allow us to build up a compositional analyzer of multithreaded programs.

4. ARCHITECTURE

Figure 2 depicts the overall structure of Checkmate. First of all, Checkmate extracts the control flow graph of a bytecode program. It then computes an approximation of the program's semantics, i.e., an element of the multithreaded semantics. Intuitively, this element relates each thread to the result of its abstract semantics, that is, the abstract state before and after each statement in the control flow graph. The inputs of the analysis are a memory model, an abstract numerical domain, and a property. Finally, Checkmate checks if the given property is respected by the result of the abstract semantics, and prints a list of warnings.

In this section we briefly sketch the approach adopted for the heap abstraction, and the interfaces of the three inputs of the analysis (namely: memory models, numerical domains, and properties). We sketch an example of interaction and the user interfaces as well.

DOI: 10.1002/spe

4.1. Memory Model

Checkmate implements the approach introduced in Section 3.3 to compute an over-approximation of all multithreaded executions allowed by the chosen (weak) memory model. Figure 3 shows the code of the MemoryModel interface. This interface contains two methods:

- get is used to read values from the shared memory. Its arguments are a reference, a string identifying the field to be read, the current state (also containing the call stack and the thread that executes the current read), and the statement that is used to read the value. get returns the abstract value read from the given location. Intuitively, it returns the upper bound of all values written in parallel by other threads and that can be seen according to the memory model.
- factory creates a new instance of the current MemoryModel. Its arguments are an object of type MultiThreadResult (the class that represents elements of our abstract thread-partitioning trace semantics) and the current iteration number of the multithreaded fixpoint semantics. factory returns an object of type MemoryModel which provides the values written in parallel contained in the given state of the thread-partitioning abstract domain.

The happens-before memory model is provided, implementing this interface. In addition, Checkmate contains two rougher memory models. The reason for implementing them is to compare the computational overhead induced by more precise memory models. The first abstraction ignores the synchronize-with relation on monitors. The second one also abstracts away the relation that tracks when and by whom a thread is launched.

4.1.1. Limitations Both the approach developed in [8] and Checkmate only consider and track precise information on synchronization via monitors and the launch of threads. Other synchronize-with relations (e.g., volatile variable and rendezvous patterns) are not considered. In these cases, Checkmate obtains sound but rough results. We made this choice essentially for two reasons:

- The discussion about what should be or not be supported by a memory model is still ongoing. In addition, several novel memory models[42, 43] have recently appeared. Since we did not want to be bound to a specific model, we focused our attention on functionalities that are always supported by memory models.
- Tracking more and more synchronize-with relations would increase both the theoretical and practical complexity of our approach. Since Checkmate is the first generic analyzer that tracks these types of relations, we wanted to investigate how this affects the complexity of our approach on standard synchronization patterns before developing more advanced features.

We believe that Checkmate can be extended in order to support other synchronization patterns, but this would require some additional work at both the theoretical and practical levels.

4.1.2. Example Consider the analysis of the data race condition on the running example of Section 1.2. With the most approximate memory model, all assignments would be seen as written in parallel with the statements executed by the initial thread, since we do not track any synchronize-with relations. Therefore, System.a.balance = 1000 of System would be seen to be written in parallel with all statements of MyThread, and in particular with System.a.getBalance(). Since the first action is not synchronized on any monitor, Checkmate produces a false alarm signaling that there may be a data race. Using a more refined memory model (both the happens-before model and the intermediate version that considers the synchronize-with relation when a thread is launched) we can check that System.a.balance = 1000 of System cannot be executed in parallel with the statements of MyThread, so that they do not form a data race. In addition, the alias analysis discovers that the accesses performed inside the two synchronized blocks are synchronized on the same monitor, so that they cannot produce a data race.

4.2. Alias Analysis

In order to obtain an effective analysis of multithreaded Java programs, we need to precisely track (i) when two accesses to the shared memory may be on the same location, and (ii) when two threads

Prepared using speauth.cls
DOI: 10.1002/spe

```
public interface NumericalValue {
  public NumericalValue add(NumericalValue v1, NumericalValue v2);
  public NumericalValue divide(NumericalValue v1, NumericalValue v2);
  public NumericalValue multiply(NumericalValue v1, NumericalValue v2);
  public NumericalValue subtract(NumericalValue v1, NumericalValue v2);
  public NumericalValue evalConstant(int v);
  public BooleanDomain testTrue(NumericalValue v1, NumericalValue v2, ComparisonOperator c);
  public BooleanDomain testFalse(NumericalValue v1, NumericalValue v2, ComparisonOperator c);
  public NumericalValue lub(NumericalValue v1, NumericalValue v2);
  public NumericalValue widening(NumericalValue v1, NumericalValue v2);
  public boolean lessEqual(NumericalValue v);
  public String  toString ();
  public boolean equals(Object v);
  public int  hashCode();
}
```

Figure 4. The code of the NumericalValue interface

are always synchronized on the same monitor. In Java the shared memory is the heap. It relates references to objects. Monitors are associated with objects, and so they are identified by reference. In addition, threads are objects, and so they are identified by reference as well. In this context, alias analysis (i.e., the way in which we abstract references) is the critical point of our analysis. In particular, we need to precisely check (i) when two references always point to the same location (must-aliasing), and (ii) when two references may point to the same location (may-aliasing). In [9] we present a combination of the must- and may- aliasing analysis. Checkmate adopts this approach.

The may-alias domain approximates all concrete references in a finite way. Intuitively, it binds each abstract reference to the program point that creates the address. Since the number of statements is bounded, this domain is composed by a finite number of elements. An abstract reference may approximate many concrete references, e.g., when a new statement is inside a loop. If two abstract references are equal, they may alias the same concrete address. If they are different, they never point to the same location. The information inferred by the may-alias analysis is also used to approximate threads, since these are objects, and so are dynamically allocated in the heap. It allows one to build up the interprocedural control flow graph, as it soundly approximates all concrete references through which a method may be dynamically invoked. In addition, we define an equivalence relation on abstract references using equivalence classes. If two abstract references point to the same equivalence class, they are equal in all possible executions, that is, they must alias the same location. This information is particularly useful to check if two threads are always synchronized on a common monitor.

4.2.1. Limitations A design choice which has been made in Checkmate is to rely on a fixed alias analysis instead of having it as a parameter. Since references are involved and are fundamental for all of the main multithreaded aspects (e.g., to identify monitors), it is important to rely on a precise and specific abstraction of the heap. In particular, we need sound information about both may- and must- aliasing. Therefore, we chose a fixed alias analysis to achieve both precision and efficiency in Checkmate.

4.3. Numerical Domain

Figure 4 shows the code of the NumericalValue interface. This interface contains some arithmetical operators (add, multiply, ...), the evaluation of conditions (testTrue and testFalse), and the common operators on lattices (lessEqual, lub, and widening). We implemented some well-known non-relational abstract domains (namely, Sign [3], Interval [3], Parity [4], and Congruence [44]).

4.3.1. Example We analyze the running example with the Sign domain. We can check that the balance of the bank account is positive (+), but we cannot precisely analyze the condition

Prepared using speauth.cls
DOI: 10.1002/spe

```
public interface Property {
  public Alert  check(MultiThreadResult r);
}

public class SingleStatementProperty implements Property {
  public SingleStatementProperty(Visitor v)  {...}
}

public interface  Visitor {
  public void checkSingleState(JVMState st, Alert a, Reference tid,
                               Statement statement, Stack<ProgramCounter> callStack);
}
```

Figure 5. The code of the Property and Visitor interfaces and of the SingleStatementProperty class

temp < 100 in MyThread, since + may be < 100. So we conclude that null may be assigned to System.a. This write action is propagated, and so System.a.printBalance() in System may cause a NullPointerException. This happens because the numerical domain is too approximate. If we use the Interval domain, we can check that the value written by System is [1000..1000]. So the condition if(temp < 100) cannot be true, null cannot be assigned to the field System.a, and the NullPointerException cannot be thrown.

4.4. Property

Figure 5 reports the main interfaces and classes dealing with the checking of properties. In particular, the Property interface defines a method check that, given a state of the thread-partitioning trace domain, returns an object of type Alert. This contains all warnings produced while checking the property. In many cases, it is not necessary to deal with the results of the abstract semantics as a whole, but it is enough to consider each statement alone. For this reason, Checkmate includes a class SingleStatementProperty that implements Property. The constructor of this class receives an object of type Visitor. This interface defines method checkSingleStatement, that receives a state, an object Alert, a thread identifier, the analyzed statement, and the call stack, and it returns an Alert object containing possible warnings.

Checkmate implements several properties. Some of them (namely, division by zero, null pointer access, and overflow) are also interesting for single-threaded code, whilst others (namely, data races, deadlock on monitors, and determinism and weak determinism as defined in [14]) are specific to parallel programs.

4.4.1. Example Considering the example presented in Section 1.2, we want to check if a data race may happen, and if a NullPointerException may be thrown. As Checkmate is parameterized by the property to be checked, we can build up an abstraction of its multithreaded executions, and we can check both properties on this abstraction.

4.5. An Example Interaction

Figure 6 depicts a UML sequence diagram that sketches one possible execution of Checkmate. The analyzer receives a memory model and a numerical domain when the analysis is started. During the analysis, Checkmate uses the memory model to know which values written in parallel are visible at a given point of execution, and the numerical domain to approximate numerical information. Once a fixpoint is reached, the analysis gets an object representing the abstraction of all possible executions of the program. Then this abstract result is passed to a Property object that checks if the property is validated. The memory model and the numerical domain are used during this phase as well. Finally, Checkmate gets an object of type Alert containing all warnings produced while checking the property; it displays this information, and it ends the analysis.

DOI: 10.1002/spe

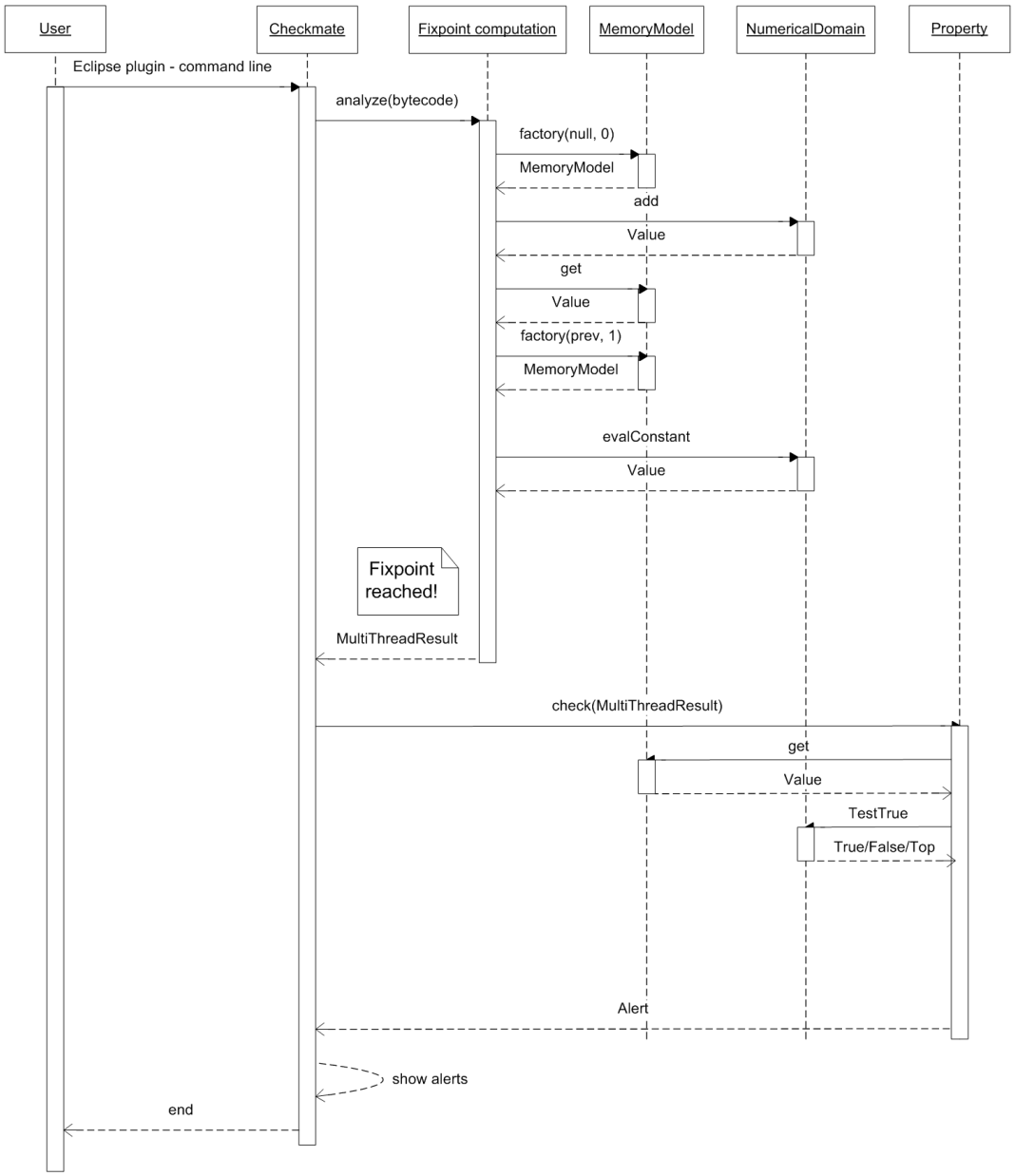


Figure 6. An example of interaction during the analysis

4.6. *User Interfaces*

We implemented two user interfaces: a command line tool, and an Eclipse plugin.

The command line tool receives all parameters of the analysis (namely, the memory model, the numerical domain, and the property) when launching the analysis. In addition, the user has to specify the directory that contains the bytecode files to analyze, and the class containing the main method. At the end of the analysis, a list of warnings or a message stating that the program is correct are printed.

The Eclipse plugin is composed of a single `.jar` file. In order to start the analysis, the user has to choose a class in the package explorer window, and click on “Checkmate” as shown in Figure 7. Then a dialog will ask the user to select the property of interest (Figure 8). At the end of the analysis, the results are displayed in a view (Figure 9). In addition, the users can set the memory

Prepared using speauth.cls
DOI: 10.1002/spe

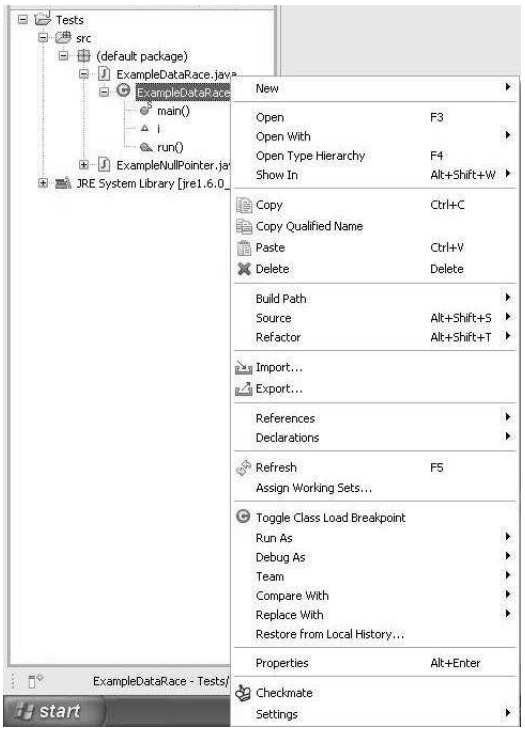


Figure 7. Launching the analysis

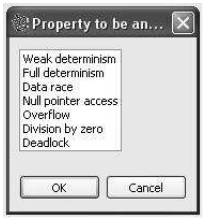


Figure 8. Choosing the property

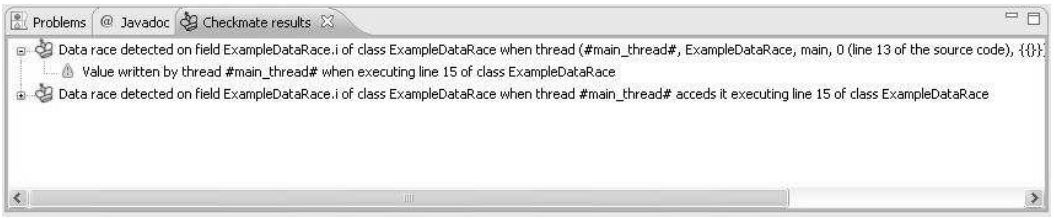


Figure 9. Output

model and the numerical domain. The default values are the happens-before memory model and the Interval domain.

5. EXPERIMENTAL RESULTS

We have applied Checkmate to several patterns of multithreading [45], case studies of weak memory models [46], an ad-hoc application to study performance, and many benchmarks [47, 45]. We investigated both the precision and the performance of the analysis. We executed Checkmate on an Intel Pentium D 3.0 Ghz with 2 GB of RAM, running Windows Server 2003 with Java virtual machine version 1.6.0_06.

5.1. Common Patterns of Multithreaded Programs

Lea [45] presents an overview of several representative patterns of Java multithreaded programs. The author shows the errors that may arise in these examples and how they can be fixed. We applied

Prepared using speauth.cls
DOI: 10.1002/spe

Checkmate to these examples[‡] to discover the errors. Since Checkmate performs a whole-program analysis, we developed a main method that exposes the behavior of interest for each example.

ExpandableArray: This class implements an array that is automatically expanded if the user wants to append an object when the array is full. All methods are synchronized. If, in parallel, a user performs two writes or a read and a write, then a conflict arises. In fact, even though all methods are synchronized, the position of the elements in the array may be non-deterministic because of threads' interleavings. This program does not contain data races, and Checkmate discovers that, while exposing the conflicts by checking the property of determinism.

LinkedCell: This class implements a list of double values. The methods that read and write the value contained in the current cell are both synchronized. The method that returns the sum of all cells is not synchronized but it relies on synchronized methods, and so it does not expose any data races. Finally, a method performs an incorrect sum, reading (without synchronization) the value contained in the first element of the list. Checkmate discovers that the well-synchronized method does not expose any data races if executed in parallel with writes to the list. It also discovers that the unsynchronized sum calculation causes a data race. If we analyze the property of determinism, we discover the well-synchronized program may produce non-deterministic behaviors.

Document: This class implements a document containing a pointer to an enclosed document. A synchronized print method is provided. Another synchronized printAll method prints all of the content using the synchronized print method of the current object, and then invokes the same method on the enclosed document. Suppose now that we have two documents d1 and d2 which are each others enclosed documents. If we print these two documents concurrently, this may cause a deadlock. For instance the first thread may start the execution of printAll and acquire the monitor of d1 starting the execution of printAll. Then the control may switch to the second thread, that acquires the monitor of d2 and then waits on the monitor of d1. Finally, the control may switch to the first thread, that starts waiting on the monitor of d2, causing a deadlock. Checkmate precisely discovers that this program may cause a deadlock.

Dot: This class implements a dot in a Cartesian plane. Its coordinates are stored in a Point object. The getter methods of the Point class are not synchronized, but all methods of class Dot are synchronized. If we move a point and shift its x value concurrently, we may obtain non-deterministic executions. Checkmate proves that this program does not contain any data race condition. In addition, it discovers the non-deterministic behaviors by checking the property of determinism.

Cell: This class implements a cell containing an integer value. The get and set methods are both synchronized. In addition, another synchronized method allows swapping the content of the current object with the content of an object passed to the method, using the getter and setter methods. If we swap the contents of two cells twice, in parallel, we may obtain a deadlock. Checkmate detects this behavior precisely checking the deadlock property.

TwoLockQueue: This class implements a queue with methods to take and put objects. If we execute a take and a put action in parallel when the queue is empty, the take action may return a null value, as it may be executed before the put action. Checkmate precisely discovers that. In particular, if the queue is empty when the two threads are executed in parallel, it signals that the value returned by the take action may be null. If we add an element before launching the two actions in parallel, Checkmate precisely discovers that the value returned by the take action cannot be null.

Account: This example is quite complex and involves many classes. In particular, it implements an immutable and a mutable account, an account holder, and two account recorders (one correct and the other one malicious). We refer the interested reader to [45] for more details about the implementation of these classes. The potential problem is that if the account holder accepts money without using an immutable instance of the recorder, a malicious recorder may cause a non-deterministic behavior. Checkmate precisely signals it this by checking the property of determinism. In addition, if the account recorder is not malicious or the account holder uses an immutable instance of the recorder, it proves that the program is deterministic.

[‡]The source code of these examples can be downloaded at `http://www.pietro.ferrara.name/checkmate/LeaExamples.zip`

DOI: 10.1002/spe

5.1.1. Discussion ℂheckmate performs a precise and correct analysis of the representative set of examples we chose from [45]. In particular, it always discovers the bug or proves that the program is correct. This result is achieved thanks to the flexibility of ℂheckmate. Using different properties allows us to tune the analysis to catch all bugs in the examples without producing false alarms. In addition, we found out that the property of determinism is often the only way to discover the behavior of interest. The focus of this property is to identify the non-deterministic behaviors due to the random interleaving of parallel threads. During the development of this property, the idea was to define a property more flexible than the data race condition to precisely identify some unwanted behaviors of multithreaded programs. Our experiments confirm that this property could overcome the limits of existing properties in some contexts. Concerning the performance, all examples are analyzed by ℂheckmate in less than a second.

5.2. Weak Memory Model

In this section, we take some challenging examples presented in [46] to test the precision of ℂheckmate. Figure 10 shows these examples. We wrote them in Java (i.e., adding a method main that instantiates and launches the threads), we compiled them with javac, and we analyzed the bytecode with ℂheckmate using the happens-before memory model and the Interval domain.

Figure 10a: A compiler may switch the statements of each thread. In fact they work on disjoint sets of variables, and so they are independent. ℂheckmate correctly tracks this behavior, and it infers that r1, r2, and r3 may be equal to zero at the end of the execution.

Figure 10b: In order to obtain the required behavior, it seems that a thread may write a variable before it reads it. Instead, this behavior may be exposed by some compiler optimizations as pointed out in Section 2.2.2 of [46]. Our analysis soundly approximates it. This behavior is exposed after the third iteration of the multithreaded semantics as (i) the first iteration writes 1 to x, (ii) at the second iteration this value is written by Thread1 to r1 and then to y, (iii) at the third iteration 1 is read by Thread2 through y and written to r2, and (iv) during the fourth and last iteration the analysis does not expose any new behavior and so it converges.

Figure 10c: Thanks to the Interval domain, ℂheckmate precisely tracks that only [0..0] can be assigned to r1 and r2. As our analysis is context-sensitive, it checks that the conditions of both threads cannot evaluate to true, and so that value 42 will never be assigned.

Figure 10d: We need three iterations of the analysis to propagate the value 1. The first iteration writes 1 to x, the second propagates it to r1 and y, and finally the third iteration assigns it to r2. In this way we obtain the result required by the example.

Figure 10e: Value 42 is assigned to x and y by Thread1. It is then assigned to x by Thread2 in parallel. Finally we come back to the first statement of Thread1 that assigns to r3 the value contained by x. In this way we capture the behavior of interest.

Figure 10f: As this example involves 4 threads, it requires more iterations of the multithreaded semantics to reach a fixpoint. ℂheckmate soundly discovers that a possible behavior yields the values $r1 == 0, r1 == r2 == 42$.

Figure 10g: This example is similar to the one in Figure 10c. The Interval domain is used to precisely infer that the condition of the loops cannot evaluate to false. Then value 42 is never assigned, neither to x nor to y, and so the threads never exit the loops. ℂheckmate discovers this fact.

Figure 10h: The situation is similar to the one in Figure 10e. ℂheckmate is precise w.r.t. the expected behavior.

5.2.1. Discussion ℂheckmate analyzes all examples successfully, producing a sound abstraction of the behavior of interest. As the figures depict toy examples (usually no more than 200 bytecode statements and 4 threads), the analysis requires always less than a second. These results are quite encouraging. We deal with examples aimed at explaining the main features of the Java memory model, and this is more restrictive than the happens-before memory model. This does not affect the precision of ℂheckmate since it successfully analyzes all examples. In general, our analysis is able to catch the behaviors presented by examples in [46] in all cases in which they do not involve

Prepared using speauth.cls
DOI: 10.1002/spe

Thread1	Thread2

r1 = x;
y = 1;
r2 = x;
	x = 1;
	r3 = y;

(a) Figure 1. Initially, x == y == 0. r1 == r2 == r3 == 0 is a legal behavior

Thread1	Thread2

r1 = x;
y = r1;
	r2 = y;
	r3 = r2|1;
	x = r3;

(b) Figure 3. Initially, x == y == 0. r1 == r2 == r3 == 1 is a legal behavior.

Thread1	Thread2

r1 = x;
if(r1! = 0)
y = 42;
	r2 = y;
	if(r2! = 0)
	x = 42;

(c) Figure 4. Initially, x == y == 0. Correctly synchronized, so r1 == r2 == 0 is the only legal behavior.

Thread1	Thread2

r1 = x;
if(r1 == 1)
y = 1;
	r2 = y;
	if(r2 == 1)
	x = 1;
	if(r2 == 0)
	x = 1;

(d) Figure 7. Initially, x == y == 0. r1 == r2 == 1 is a legal behavior.

Thread1	Thread2

r3 = x;
if(r3 == 0)
x = 42;
r1 = x;
y = r1;
	r2 = y;
	x = r2;

(e) Figure 11. Initially, x == y == z == 0. r1 == r2 == r3 == 42 is a legal behavior.

Thread1	Thread2	Thread3	Thread4
r1 = x; y = r1;	r2 = y; x = r2;	z = 42;	r0 = z; x = r0;

(f) Figure 12. Initially, x == y == z == 0. r0 == 0, r1 == r2 == 42 is a legal behavior.

Thread1	Thread2

do{
r1 = x;
}while(r1 == 0);
y = 42;
	do{
	r2 = y;
	}while(r2 == 0);
	x = 42;

(g) Figure 25. Initially, x == y == 0. Correctly synchronized, so non-termination is the only legal behavior

Thread1	Thread2

r1 = x;
if(r1 == 0)
x = 1;
r2 = x;
y = r2;
	r3 = y;
	x = r3;

(h) Figure 27. Initially, x == y == 0. Compiler transformations can result in r1 == r2 == r3 == 1.

Figure 10. Some examples taken from [46]

volatile variables. In the remaining cases, our analysis does not take into account the fact that a variable is volatile. For this reason, we obtain results that are still sound but too approximate. On the other hand, we believe our framework to be extensible and flexible enough to also take into account volatile variables, and this could be done by implementing another memory model.

5.3. *Examples of Increasing Sizes*

We applied Checkmate to several examples of increasing size that simulate the operations performed by a bank[§]. Figure 11 reports the number of abstract threads and statements of each program. Figure 12 reports the time taken by the analysis (in msec) to build up the abstraction of the

[§]The source code of these examples can be downloaded at `http://www.pietro.ferrara.name/checkmate/Incremental.zip`

Prepared using speauth.cls
DOI: 10.1002/spe

Program	# ab. th.	# st.
Test1	3	452
Test2	5	684
Test3	7	807
Test4	11	1049
Test5	13	1173
Test6	15	1405
Test7	17	1526
Test8	19	1758
Test9	20	1878
Test10	24	2294

Figure 11. Number of abstract threads and statements

Program	Top	Sign	Interval	Parity	Congruence
Test1	814	361	217	404	294
Test2	409	391	356	620	545
Test3	712	595	925	521	642
Test4	799	823	3806	703	642
Test5	1090	919	5887	779	616
Test6	1382	824	7161	900	986
Test7	1071	1647	9289	1340	863
Test8	1018	1269	10999	1263	1221
Test9	1421	2212	11691	1274	1623
Test10	1466	2432	17016	863	1906

Figure 12. Analysis Time (msec) vs Domain

Program	Weak det.	Det.	Data race	Null	Overflow	Div. by 0	Deadlock
Test1	31	32	47	31	15	16	16
Test2	78	78	125	47	47	47	15
Test3	125	125	172	187	63	62	16
Test4	250	250	359	125	94	94	62
Test5	359	360	484	172	125	125	78
Test6	547	562	828	266	219	203	125
Test7	719	734	1047	313	250	250	156
Test8	1000	1047	1609	438	359	360	250
Test9	1203	1234	1938	516	406	422	265
Test10	2031	2094	3609	828	688	687	500

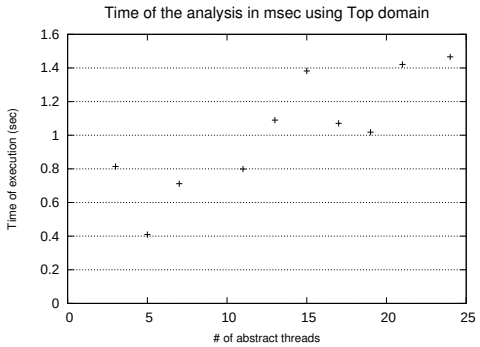
Figure 13. Analysis Time (msec) vs Property

program using different numerical domains. The analyzer is quite fast: it rarely requires more than a couple of seconds to converge. Only the Interval domain requires more time (about 17 seconds in the worst case), as it is the most complex domain that we implemented.

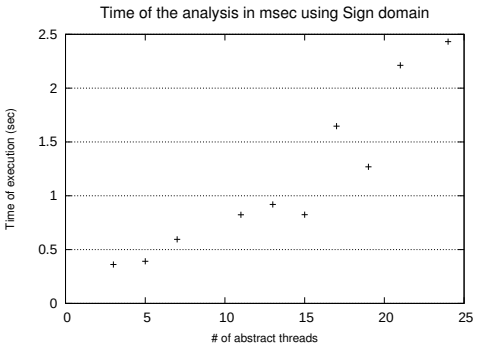
We want to study the complexity of the analysis w.r.t. the number of statements and abstract threads. For each numerical domain, we plot the number of abstract threads (in the x-axis), against the execution time (in the y-axis). The behavior of the Top domain (Figure 14a) is not regular. The execution times grow but, as the analysis is quite fast, it is hard to conclude how it grows exactly. More regular results are obtained with Parity (Figure 14d), Sign (Figure 14b), and Congruence (Figure 14e). The complexity is linear w.r.t. the number of abstract threads in all cases. Finally, the Interval domain (Figure 14c) seems to expose a quadratic complexity. We also investigated how the analysis time varies w.r.t. the number of abstract threads analyzed. Figure 14f plots the execution time per thread. All domains except Interval require a constant time per thread. In the case of the Interval domain, the times per thread increase w.r.t. the number of abstract threads. This increase seems to be linear, and this confirms that the overall time required by the analysis is quadratic w.r.t. the number of abstract threads.

Starting from these experimental results, we conclude that the complexity exposed by Checkmate in practice is quadratic w.r.t. the number of threads and statements. This result is promising, but we

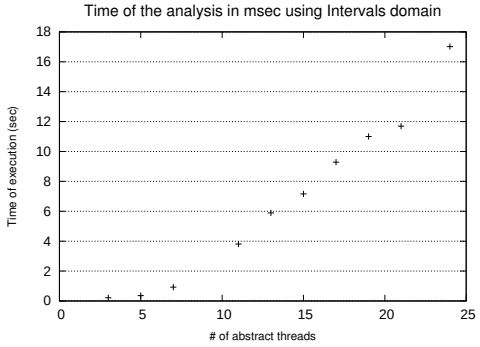
Prepared using speauth.cls
DOI: 10.1002/spe



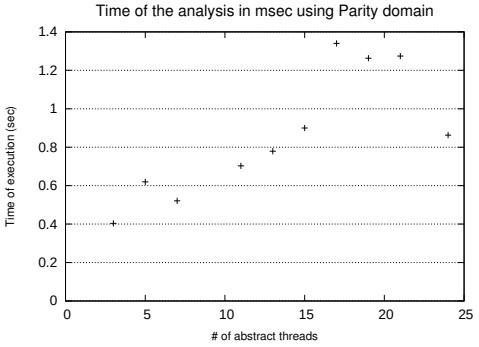
(a) Top domain



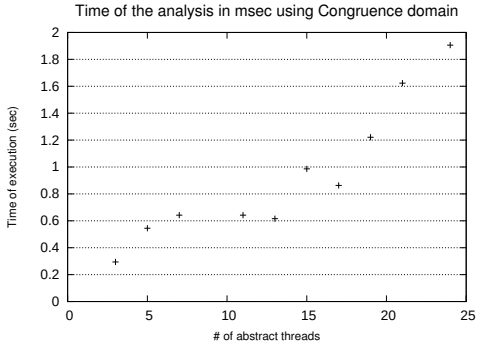
(b) Sign domain



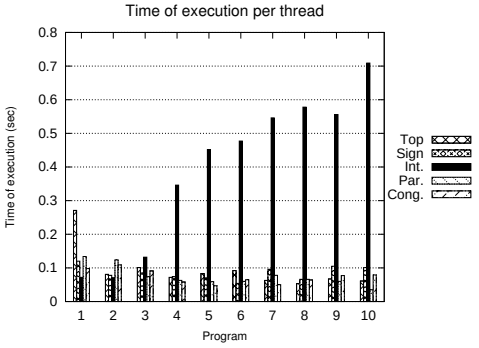
(c) Interval domain



(d) Parity domain



(e) Congruence domain



(f) Per thread

Figure 14. Times of execution

think that the analysis could be optimized. In particular the fixpoint computation of single-thread semantics is sometimes slow as it works at a low abstraction level by simulating step by step the actions of the Java virtual machine.

DOI: 10.1002/spe

Program	# st.	# ab. th.
philo	213	2
forkjoin	170	2
barrier	363	3
sync	320	3
crypt	2636	3
sor	1121	2
elevator	1829	2
lufact	3732	2
montecarlo	3864	2
total	14248	21

Figure 15. The analyzed programs

5.4. Benchmarks

In this section, we apply Checkmate to the analysis of some benchmarks adopted to evaluate the performances of static analyses of multithreaded programs. Two applications (philo, and elevator) are taken from [47], while the others (barrier, forkjoin, sync, sor, crypt, lufact, and montecarlo) are taken from the Java Grande Forum Benchmark Suite [48]. We removed from the original programs the calls to system functions (e.g., System.out.println) as sometimes they deal with native methods or reflection that are not supported by Checkmate. Figure 15 reports the analyzed programs, the number of statements, and the number of abstract threads. Note that in all cases the abstract threads approximate a potentially unbounded number of concrete threads. We apply the analysis to all benchmarks with all possible combinations of memory models and abstract numerical domains. Figure 16 reports the computational times. For each numerical domain we report the times of execution using (i) the most relaxed memory model (column **AP**), (ii) the memory model that tracks only when a thread is launched (column **TL**), and (iii) the happens-before memory model (column **HB**). In addition, column **S.T.** reports the time spent to compute the semantics of each thread in isolation.

For each numerical domain, we plot the times of the analysis using the three memory models, against the overall number of analyzed statements. Figure 18a reports the result obtained applying the Top domain, while Figures 18b-e do the same with the Sign, Interval, Parity and Congruence domains, respectively. In all cases, the analysis is quite fast for programs with fewer than 500 statements. In addition, we observe comparable computational times for same program using different numerical domains. The analysis of crypt is always quite faster than that of elevator, even though it is larger. This happens because of the internal structure of the program. With the exception of the Interval and (in part) the Sign domains, the time of the analysis does not grow considerably w.r.t. the number of statements. The complexity seems to be almost linear. Instead, Interval and in part Sign seem to expose a higher complexity.

The analysis of montecarlo is notably slower. We wanted to check if this slowness is due to our approach or to the fixpoint computation of a single thread, i.e., to the structure of the program. Figure 17 reports the overhead due to the multithreaded fixpoint computation compared with the single-threaded fixpoint semantics. Figure 18f depicts this overhead when applying the Interval domain and the happens-before memory model. It makes clear that this overhead does not depend on the number of threads or statements analyzed. Its values are between 250% and 450% (with the exception of crypt), and do not depend on the program size.

In fact, we often obtain the greatest overhead for the smallest application. In addition, for larger applications (sor, elevator, lufact, and montecarlo) the overhead is almost stable (300% in average). This result is quite encouraging: it means that on average we need about 3 iterations of the single-threaded semantics to reach a fixpoint in our multithreaded semantics. In addition, we think that we can improve this result, since our implementation is not optimized at all. For instance, we may parallelize the analysis of different threads during the same iteration of the multithreaded semantics.

DOI: 10.1002/spe

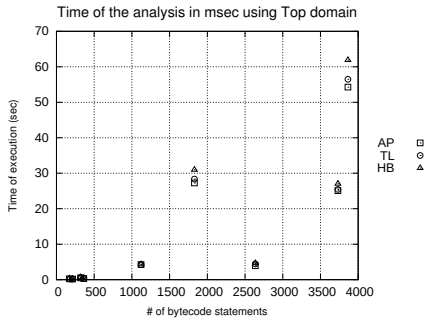
Program	Top AP	Top TL	Top HB	Top S.T.	Sign AP	Sign TL	Sign HB	Sign S.T.	Int. AP	Int. TL	Int. HB	Int. S.T.	Par. AP	Par. TL	Par. HB	Par. S.T.	Cong. AP	Cong. TL	Cong. HB	Cong. S.T.
philo	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”
forkjoin	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”
barrier	<1”	<1”	<1”	<1”	<1”	1”	1”	<1”	1”	1”	2”	1”	<1”	1”	1”	<1”	1”	1”	1”	<1”
sync	1”	1”	1”	<1”	1”	1”	1”	<1”	2”	2”	3”	1”	1”	1”	1”	<1”	1”	1”	2”	1”
crypt	4”	4”	5”	1”	6”	6”	6”	3”	16”	17”	17”	13”	5”	5”	6”	1”	4”	5”	5”	2”
sor	4”	4”	4”	2”	6”	6”	7”	2”	16”	17”	17”	5”	5”	5”	6”	2”	5”	5”	5”	2”
elevator	27”	28”	31”	11”	10”	11”	11”	4”	18”	18”	19”	7”	29”	30”	30”	11”	28”	29”	29”	11”
lufact	25”	25”	27”	10”	52”	52”	53”	20”	5’52”	5’56”	5’59”	2’08”	28”	29”	29”	12”	27”	29”	29”	11”
montecarlo	54”	56”	1’02”	23”	2’23”	2’26”	2’35”	45”	1h00’33”	1h00’38”	1h00’56”	16’48”	1’38”	1’38”	1’43”	31”	54”	1’00”	1’04”	25”

Figure 16. Times of analysis

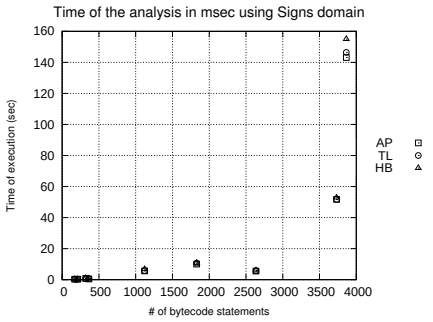
Program	Top AP	Top TL	Top HB	Sign AP	Sign TL	Sign HB	Int. AP	Int. TL	Int. HB	Par. AP	Par. TL	Par. HB	Cong. AP	Cong. TL	Cong. HB	Total AP	Total TL	Total HB
philo	221%	263%	282%	245%	246%	259%	450%	480%	532%	145%	198%	198%	160%	297%	319%	233%	282%	300%
forkjoin	352%	403%	531%	282%	315%	332%	279%	314%	317%	422%	319%	559%	432%	454%	495%	348%	380%	434%
barrier	148%	193%	207%	173%	212%	255%	193%	205%	224%	182%	199%	218%	275%	278%	299%	193%	215%	236%
sync	233%	296%	310%	316%	323%	369%	398%	400%	435%	268%	300%	307%	228%	242%	282%	295%	309%	343%
crypt	295%	334%	349%	201%	208%	214%	120%	127%	128%	366%	341%	395%	258%	277%	281%	171%	181%	186%
sor	263%	261%	267%	252%	260%	301%	304%	324%	325%	225%	265%	247%	251%	256%	264%	269%	282%	292%
elevator	240%	249%	272%	234%	248%	252%	243%	252%	258%	256%	253%	266%	244%	251%	252%	245%	254%	262%
lufact	259%	262%	279%	252%	253%	258%	274%	277%	279%	230%	267%	238%	235%	249%	254%	265%	269%	272%
montecarlo	235%	245%	268%	320%	328%	347%	361%	361%	363%	311%	295%	328%	212%	235%	251%	352%	353%	357%
average	249%	279%	307%	253%	266%	287%	291%	304%	318%	267%	271%	306%	255%	282%	300%			

Figure 17. Overhead of multithreaded fixpoint computation

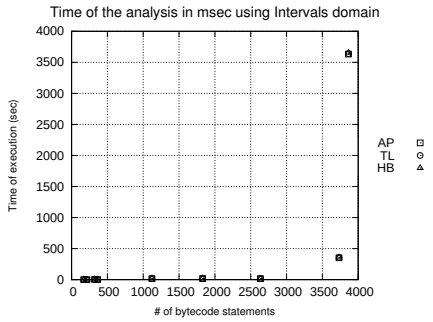
Prepared using speauth.cls
DOI: 10.1002/spe



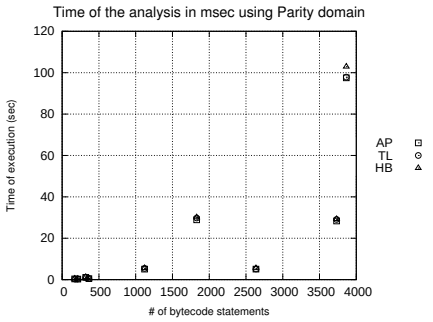
(a) Top domain



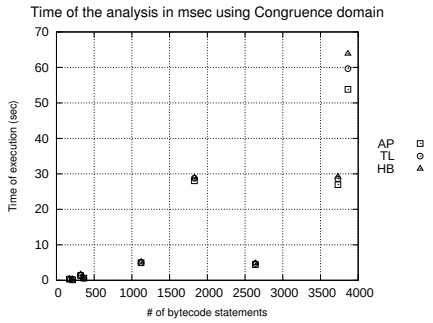
(b) Sign domain



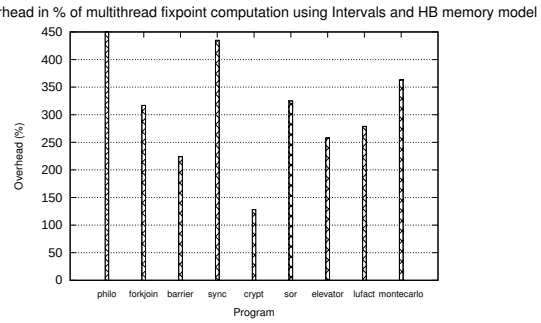
(c) Interval domain



(d) Parity domain



(e) Congruence domain



(f) Overhead of multithreaded fixpoint computation using Interval

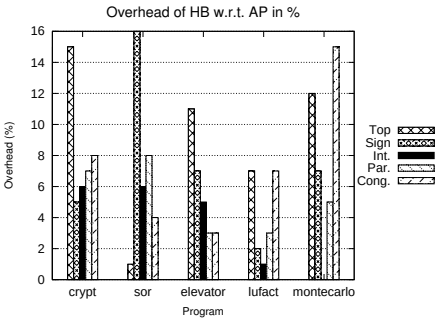
Figure 18. Times of execution using HB memory model

Finally, we compared the times of the analysis using different memory models. For programs with fewer than 500 statements, the analysis is too fast to obtain significant comparisons. So we consider only the analyses of crypt, sor, elevator, lufact, and montecarlo. Figure 19a depicts the overhead of the analysis using the happens-before memory model against the AP memory model, while Figure 19b depicts the overhead compared with the TL memory model. The overhead of HB compared with AP is rarely greater than 10%, and on average it is about 5%. It is on average about 2% w.r.t. TL and rarely greater than 5%. We believe that these results are quite encouraging as well: the overhead of more refined memory models is quite small. This means that tracking more and more relations between threads does not affect the performance of the analysis significantly.

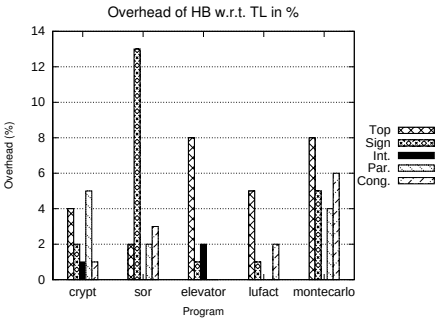
5.5. Comparison

Figure 20 summarizes the main achievements of some representative existing tools and Checkmate. Column **Sound** reports whether or not the analyzer is sound w.r.t. all multithreaded executions. We identified two levels of **Efficiency**: **Fast** means that the analyzer can be applied to few thousands of

Prepared using speauth.cls
DOI: 10.1002/spe



(a) Overhead of HB w.r.t. AP



(b) Overhead of HB w.r.t. AP

Figure 19. The overhead in % of the HB memory model

Analysis	Sound	Efficiency		Accuracy		Flexible	Automatic
		Fast	Scalable	Precise	Complete		
Checkmate	✓	✓	✗	✓	✗	✓	✓
Clousot[5]	✗	✓	✓	✓	✗	✓	✗
Chess [16]	✗	✗	✗	✓	✓	✗	✓
Abadi et al. [21]	✓	✓	✓	✓	✗	✗	✗
Naik and Aiken [22]	✓	✗	✗	✓	✗	✗	✓
Williams et al. [28]	✗	✓	✓	✓	✗	✗	✓

Figure 20. Comparison with other tools

LOC in a reasonable (that is, within a few minutes) time, while **Scalable** means that it can be applied to hundreds of thousands of LOC. Concerning the **Accuracy** of the analysis, we distinguished between **Precise** (that is, it produces few false alarms in practice) and **Complete** (that is, it produces no false alarms). Column **Flexible** reports whether or not the analyzer can be applied to various properties, while column **Automatic** reports whether or not the analyzer requires any form of manual annotation (e.g., types or contracts).

The tool that is most similar to Checkmate is Clousot. The main differences w.r.t. Checkmate are that Clousot does not consider multithreaded executions, it requires manual annotation, while on the other hand it achieves scalability. A quite different tool is Chess: it considers multithreaded programs but it is not sound since it binds the number of context switches to a constant value. On the other hand, it is the only tool that achieves completeness, since it executes the program. For this reason, Chess is not efficient, but it is completely automatic. Finally, it is not flexible since although it can be tuned to a specific property, it always checks all runtime errors. None of the other three analyzers are flexible, since they check a specific multithreaded property (data races or deadlocks). The type system proposed by Abadi et al. scales, but it requires manual annotation of the program with extra type information. The must alias analysis proposed by Naik and Aiken is lacking in terms of efficiency, while the deadlock checking proposed by Williams et al. is particularly efficient, but it is not sound w.r.t. all possible multithreaded executions.

Given this context, Checkmate represents a unique result. In particular, it is the first flexible analyzer that is sound for multithreaded programs.

DOI: 10.1002/spe

5.6. *Limitations*

The experimental results underline both the efficiency and the precision of Checkmate, but they prove that the analysis does not scale to industrial software as well. Since the analysis we perform is whole-program, we are not in a position to analyze a complete industrial application in a reasonable time. The current generic analyzers that scale (e.g., `Clousot`) reason modularly about methods, relying on contracts when methods are invoked. Unluckily, precise modular reasoning is not possible on multithreaded programs without imposing further restrictions on programs (e.g., absence of data races) and applying a non-standard specification methodology. The goal of Checkmate was to (i) analyze all multithreaded Java programs, and not to restrict programs to avoid data races and deadlocks, and (ii) check properties for all possible executions without requiring additional automatic synchronizations.

In this context, we had to develop a whole-program analysis. On the other hand, it seems evident that something more is required to put developers in a position to think modularly about threads. For instance, a different approach has been adopted by Software Transactional Memory [49]. However, STM has not been adopted by common programming languages (e.g., Java and C#) which, so far, offer threads rather than transactions.

Concerning precision, while Checkmate is able to deal effectively with many case studies, we could still improve its precision in three main directions:

- supporting more synchronization patterns, as we pointed out in Section 4.1,
- defining and implementing specific abstract domains for the properties of interest that do not deal with concurrency, and
- manually developing the semantics of some native methods.

6. CONCLUSION

We have presented Checkmate, a generic static analyzer based on abstract interpretation for multithreaded Java programs. It supports the most relevant features of the Java language, such as unbounded dynamic thread creation, runtime creation and management of monitors, method-calls in the presence of overloading, overriding, and recursion, and dynamic allocation of shared memory. We presented the overall structure of the analyzer, and we studied in detail the experimental results obtained when applying Checkmate to some common patterns of concurrent programming in Java [45], to some case studies about the Java memory model presented in [46], to an incremental application, and to a set of well-known benchmarks [47, 48]. The precision exposed when analyzing these examples is quite encouraging, and we are in a position to analyze programs with an unbounded number of threads and thousands of statements in a limited time.

Future work concerns the refinement of the analysis and its application to industrial software. In particular, we want to refine the numerical domain to apply relational analyses. In order to reach this goal, we need to perform some transformations on the bytecode, e.g., stack abstraction and expression recovery [50]. Our aim is also to refine our memory model to track more synchronization actions, and with some restrictions of the Java memory model that are not considered by the happens-before memory model.

REFERENCES

1. Lee EA. The problem with threads. *Computer*, IEEE Computer Society Press, 2006.
2. Sutter H, Larus J. Software and the concurrency revolution. *ACM Queue*, ACM Press, 2005.
3. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of POPL '77*, ACM, 1977.
4. Cousot P, Cousot R. Systematic design of program analysis frameworks. *Proceedings of POPL '79*, ACM, 1979.
5. Logozzo F, Fähndrich M. Static contract checking with abstract interpretation. *Proceedings of FoVeOOS '10*, LNCS, Springer-Verlag, 2010.
6. Logozzo F. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. *Proceedings of VMCAI '07*, LNCS, Springer-Verlag, 2007.

DOI: 10.1002/spe

7. Spoto F. The Julia Generic Static Analyser. http://profs.sci.univr.it/~spoto/julia/.
8. Ferrara P. Static analysis via abstract interpretation of the happens-before memory model. *Proceedings of TAP '08*, LNCS, Springer-Verlag, 2008.
9. Ferrara P. A fast and precise analysis for data race detection. *Bytecode '08*, 2008.
10. Ferrara P. Static analysis of the determinism of multithreaded programs. *Proceedings of SEFM '08*, IEEE Computer Society, 2008.
11. Lindholm T, Yellin F. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.
12. Odersky M. *The Scala Language Specification* 2008.
13. Meyer B. *Object-Oriented Software Construction (2nd Edition)*. Prentice Hall, 1997.
14. Ferrara P. Checkmate: a generic static analyzer of java multithreaded programs. *Proceedings of SEFM '09*, IEEE Computer Society, 2009.
15. Qadeer S, Rehof J. Context-bounded model checking of concurrent software. *Proceedings of TACAS '05*, LNCS, Springer-Verlag, 2005.
16. Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. *Proceedings of PLDI '07*, ACM Press, 2007.
17. Ramalingam G. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.* 2000; **22**:416–430.
18. Flanagan C, Qadeer S. Thread-modular model checking. *Proceedings of SPIN '03*, Springer, 2003.
19. Qadeer S, Rajamani SK, Rehof J. Summarizing procedures in concurrent programs. *Proceedings of POPL '04*, ACM, 2004.
20. Rinard MC. Analysis of multithreaded programs. *Proceedings of SAS '01*, LNCS, Springer-Verlag, 2001.
21. Abadi M, Flanagan C, Freund SN. Types for safe locking: Static race detection for java. *Proceedings of TOPLAS '06*, ACM Press, 2006.
22. Naik M, Aiken A. Conditional must not aliasing for static race detection. *Proceedings of POPL '07*, ACM Press, 2007.
23. Kahlon V, Yang Y, Sankaranarayanan S, Gupta A. Fast and accurate static data-race detection for concurrent programs. *Proceedings of CAV '07*, LNCS, Springer-Verlag, 2007.
24. Henzinger TA, Jhala R, Majumdar R. Race checking by context inference. *Proceedings of PLSI '04*, 2004.
25. Bensalem S, Fernandez J, Havelund K, Mounier L. Confirmation of deadlock potentials detected by runtime analysis. *Proceedings of PADTAD '06*, ACM Press, 2006.
26. Bensalem S, Havelund K. Scalable dynamic deadlock analysis of multithreaded programs. *Proceedings of PADTAD '05*, ACM Press, 2005.
27. Eytani Y, Havelund K, Stoller SD, Ur S. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurr. Comput. : Pract. Exper.* 2007; **19**(3).
28. Williams A, Thies W, Ernst MD. Static deadlock detection for Java libraries. *Proceedings of ECOOP '05*, LNCS, Springer-Verlag, 2005.
29. Agarwal R, Wang L, Stoller SD. Detecting potential deadlocks with static analysis and runtime monitoring. *Proceedings of PADTAD '05*, Springer-Verlag, 2005.
30. Spoto F. Nullness analysis in boolean form. *Proceedings of SEFM '08*, IEEE Computer Society Press, 2008.
31. Fähndrich M, Leino KRM. Declaring and checking non-null types in an object-oriented language. *Proceedings of OOPSLA '03*, ACM Press, 2003.
32. Hovemeyer D, Pugh W. Finding more null pointer bugs, but not too many. *Proceedings of PASTE '07*, ACM Press, 2007.
33. Hubert L, Jensen T, Pichardie D. Semantic foundations and inference of non-null annotations. *Proceedings of FMOODS '08*, Springer-Verlag, 2008.
34. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. The ASTRÉE analyzer. *Proceedings of ESOP '05*, LNCS, Springer-Verlag, 2005.
35. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R. Extended static checking for java. *Proceedings of PLDI '02*, ACM Press, 2002.
36. Java program checker J. http://artho.com/jlint/.
37. Cousot P. The calculational design of a generic abstract interpreter. *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
38. Manson J, Pugh W, Adve SV. The Java memory model. *Proceedings of POPL '05*, ACM Press, 2005.
39. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 1979.
40. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM Press, 1978.
41. Ferrara P. Static analysis via abstract interpretation of multithreaded programs. PhD Thesis, Ecole Polytechnique of Paris (France) and University "Ca' Foscari" of Venice (Italy) May 2009.
42. Batty M, Owens S, Sarkar S, Sewell P, Weber T. Mathematizing c++ concurrency. *Proceedings of POPL '11*, Press A (ed.), 2011.
43. Owens S, Sarkar S, Sewell P. A better x86 memory model: x86-tso. *Proceedings of TPHOLs '09*, Springer (ed.), LNCS, 2009.
44. Granger P. Static analysis of linear congruence equalities among variables of a program. *Proceedings TAPSOFT '91*, LNCS, Springer-Verlag, 1991.
45. Lea D. *Concurrent Programming in Java*. Addison-Wesley, 1996.
46. Manson J, Pugh W, Adve S. The Java Memory Model. http://unladen-swallow.googlecode.com/files/journal.pdf.
47. Von Praun C, Gross TR. Object race detection. *Proceedings of OOPSLA '01*, ACM Press, 2001.
48. Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/research/activities/java-grande/.
49. Shavit N, Touitou D. Software transactional memory. *Symposium on Principles of Distributed Computing*, ACM Press, 1995.

DOI: 10.1002/spe

50. Logozzo F, Fähndrich M. On the relative completeness of bytecode analysis versus source code analysis. *Proceedings of CC '08*, LNCS, Springer-Verlag, 2008.

DOI: 10.1002/spe