Checkmate: a Generic Static Analyzer of Java Multithreaded Programs

Pietro Ferrara

ETH Zurich
pietro.ferrara@inf.ethz.ch

***Abstract*—In this paper we present Checkmate, a generic static analyzer of Java multithreaded programs based on the abstract interpretation theory. It supports all the most relevant features of Java multithreading, as dynamic unbounded thread creation, runtime creation of monitors, and dynamic allocation of shared memory. We implement a wide set of properties, from the ones interesting also for sequential programs, e.g. division by zero, to the ones typical of multithtreaded programs, e.g. data races. We analyze several external case studies and benchmarks with Checkmate, and we study the experimental results both in term of precision and efficiency. It turns out that the analysis is particularly accurate and we are in position to analyze programs composed by some thousands of statements and a potentially infinite number of threads. As far as we know, Checkmate is the first generic static analyzer of Java multithreaded programs.**

***Keywords*-Static Analysis; Abstract Interpretation; Multithreaded Programs**

I. Introduction

Testing and debugging multithreaded programs is particularly difficult [1]. Some executions may be rarely exposed during the development of an application (e.g. because they rely on a particular interleaving of threads), and, even if they are discovered, it would be difficult to reproduce them. In addition, some executions may be exposed only by specific compilers, virtual machines or hardware architectures.
On the other hand, while the improvement of single-core processors performances is slowing down, multi-core architectures appear to be the most promising way to prolong the Moore's Law [2]. Even if any application that runs on a single-core processor runs also on a multi-core one, in order to fully take advantage of these capabilities the applications must be optimized for multithreading. It is necessary to develop programs with explicit parallelism (i.e. multithreading in Java), otherwise a single application would not take relevant advantage from the multi-core hardware. In addition, many-core will be the trend in the next future.
The obvious consequence of this situation is that tools able to discover bugs in multithreaded software are particularly welcomed [3]. A priori all types of bugs are interesting: null pointers accesses, data races, divisions by zero, deadlocks, etc..
Static analysis is aimed at over-approximating the semantics of a program in order to prove a property on all the possible executions. Generic analyzers have been deeply studied during last years. The idea behind them is to develop a static analyzer that defines the semantics of statements and programs, while it can be plugged with various numerical domains, and in order to check various properties.
Some generic static analyzers based on abstract interpretation were proposed. These can be fitted with different domains (in order to obtain faster and more approximated or slower and more refined analyses) and in order to analyze different properties. Some example of these analyzers are [4], [5], [6], [7].
Existing generic analyzers do not support multithreading. If generic static analyzers have been successfully applied to the analysis of single-thread programs, their application to multithreaded programs should be at least as successful as for sequential programs, since multithreading seems to be particularly interested in tools that help to debugging.

A. Contribution

In this paper we present Checkmate[1], a generic analyzer of Java multithreaded programs based on the abstract interpretation theory [8], [9]. It is generic with respect to the numerical domain, the analyzed property, and the memory model. It is tuned at bytecode level [10], so it can analyze libraries whose source code is not available, and programs written in other languages that compile into Java bytecode, e.g. Scala [11]. Checkmate supports all the main features of the Java multithreading system, and in particular dynamic unbounded threads creation, runtime creation and management of monitors, and dynamic allocation of shared memory. In addition, it supports all common features of Java, as strings, arrays, static fields and methods, method-calls in 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 annotation, e.g. contracts [12].
We develop some well-known nonrelational numerical domains and some properties, and we apply Checkmate to the analysis of some external case studies and benchmarks in order to study the precision and the efficiency. It turns out that

[1] http://www.pietro.ferrara.name/checkmate

Checkmate is (i) precise enough in order to catch possible bugs on common patterns of multithreaded programs, (ii) fast enough in order to be applied to programs containing some thousands of statements and a potentially unbounded number of threads.
As far as we know, Checkmate is the first generic static analyzer of Java multithreaded programs.

II. Background

Checkmate is the implementation of a theoretical work developed during last years in order to apply generic analyzers to multithreaded programs [13]. In this section, we present such results briefly, and we refer the interested reader to the cited articles.

A. Abstract Interpretation

Abstract interpretation is a theory to define and soundly approximate the semantics of a program [8], [9]. Roughly, a concrete semantics, aimed at specifying the runtime properties of interest, is defined. Then it is approximated through one or more steps in order to finally obtain 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 in order to make the analysis convergent.
Abstract interpretation can be applied in order to develop generic analyzers [14]. In particular, this theory allows to define a compositional analysis, e.g. an analysis that can be instantiated with different numerical domains, and in order to analyze different properties.

B. Memory Model

Memory models define which behaviors are allowed during the execution of a multithreaded programs. In particular, they specify which values written in parallel may be read from the shared memory. The interest on this topic increased recently: for instance, the first specification of the Java Virtual Machine [10] was flawed [15], and only recent work [16] revised it. This solution is quite complex, especially from a static analysis point of view. Other memory models have been proposed in the past: Lamport [17] formalized the sequentially consistency rule. It is quite simple, but too much restrictive. A good compromise is the happens-before memory model [18]: it is an over-approximation of the Java one, and it is simple enough to base a static analysis on it. In [19] we define the happens-before memory model [2] in a way that is amenable to a fixpoint computation, and then we abstract it with a computable semantics. This approach is completely modular with respect to the semantics of statements of the programming language, the domains used in order to trace information on numerical values, references, etc.., and the property of interest. The intuition is that the core of the memory model can be defined as a function that returns the set of visible values at a given point of the execution when a thread is reading from the shared memory. In order to be sound with respect to all the possible executions, we compute two nested fixpoints. The inner one computes the semantics of a single thread given a multithreaded execution from which it extracts the values written in parallel that may be visible. The outer one iterates for all active threads their single-thread semantics until a global fixpoint is reached. Each iteration of this fixpoint computation may expose for each thread new visible values written in parallel, thus causing new executions during the following iteration of the fixpoint semantics. Iterating this process until a fixpoint is reached allows us to obtain an approximation of all the possible multithreaded executions.
Checkmate implements this approach in order to compute an over-approximation of all the multithreaded executions allowed by a memory model.

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

C. Alias Analysis

In order to obtain an effective analysis of Java multithreaded programs we need to precisely trace (i) when two accesses to the shared memory may be on the same location, (ii) when two threads are always synchronized on the same monitor.
In Java the shared memory is the heap. It relates references to objects. Monitors are defined on objects, and so they are identified by reference. In addition threads are objects, and so they are identified by reference too.
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), (ii) when two references may point to the same location (may-aliasing).
In [20] we present a combination of the must- and may-aliasing analysis. Checkmate adopts the same approach.
On one hand, the may-alias domain approximates all the concrete references in a finite way. Intuitively, it bounds 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 the new statement is inside a loop iterated many times. 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 in order to approximate threads, since these are objects, and so dynamically allocated in the heap. In addition, it allows us to build up the interprocedural control flow graph, as it soundly approximate all the concrete references through whom a method may be dynamically invoked.
On the other hand, we relate each abstract reference to an

equivalence class. If two abstract references point to the same equivalence class, they are equals in all the possible executions.

D. Deterministic Property

The most part of the properties implemented in Checkmate is well-known, e.g. data race, and division by zero. In addition, we implement two properties, i.e. determinism and weak determinism, that we introduced in [21].
The intuition behind the definition of the determinism is to trace information on the effects of unordered communications through shared memory, rather than working on the reasons that cause them, i.e. accesses in conflict. This approach is strictly more flexible than existing ones, as it can be easily restricted only on a part of the shared memory, on a subset of the active threads, on some statements, etc.. In particular, the weak determinism relaxes the full determinism allowing some nondeterministic behaviors if they produce the same abstract numerical values.

III. System Architecture

Figure 1 depicts the overall architecture of Checkmate. The first step of the analysis is to receive the source code of a Java program, and to compile it with javac (or eventually it receives directly the bytecode of the program). The bytecode is then parsed by BCEL [22], and we build up the intraprocedural control flow graph. Thus Checkmate computes an approximation of the program's semantics starting the analysis from the main method of a given class. The inputs of this part of the analysis are a memory model and an abstract numerical domain. Finally, it checks if a given property received as input is respected by the given abstraction. If it is not the case, a list of warnings is displayed.
In the following of this section we present the interfaces of the three inputs of the analysis, i.e. properties (interface Property), memory models (interface MemoryModel), and numerical domains (interface NumericalValue).

A. Property

Figure 2 depicts the interfaces to be implemented in order to check a property. Interface Property requires to define a method check that given an approximation of all the multithreaded executions returns an object of type Alert. This contains all the warnings produced checking the property. In this way, it deals directly with the complete approximation of the program's executions. This is required when we want to implement properties that need to analyze the comprehensive execution of different threads, e.g. in order to discover deadlocks and data races. In many other cases, we can consider each state of computation separately, e.g. in order to discover divisions by zero and accesses to null pointers. Class SingleStatementProperty is a shortcut

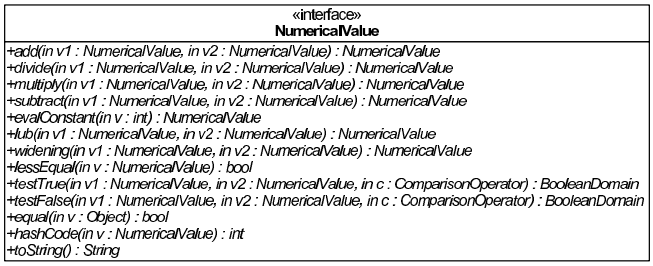

Figure 3. NumericalDomain interface

Figure 4. MemoryModel interface

in order to develop such properties. Its construct receives an object of type Visitor; this interface requires to implement a method that check locally if a state respect the property. All the warnings have to be added on an Alert object.

B. Numerical Domain

Figure 3 depicts the UML object diagram of numerical domains. Interface NumericalValue requires to implement all the abstract arithmetical operators (add, multiply, ...), the evaluation of conditions (testTrue and testFalse), and the common operators on lattices (lessEqual, lub, and widening).

C. Memory Model

Figure 4 depicts the object diagram of the interface MemoryModel. Two methods are defined on it.
get receives as parameters a reference, a string identifying the field to be read, the current state (containing also the call stack and the thread that executes the current read), and the statement that is used to read the value. It returns the value read on that locations. It takes the upper bound of all the values written in parallel by other threads and that can be seen following a memory model. factory receives as parameters an object of type MultiThreadResult (that is the class that represents approximations of all the multithreaded executions) and the number of iteration of the multithreaded fixpoint semantics. It returns an object of type MemoryModel. This has to provide the values written in parallel following the given MultiThreadResult object.

IV. Implementation

In this section we introduce the user interfaces and the main parameters we implemented in Checkmate.

A. User Interfaces

We implemented two different user interfaces: a command line tool, and an Eclipse plugin.
Command Line: The command line tool is composed

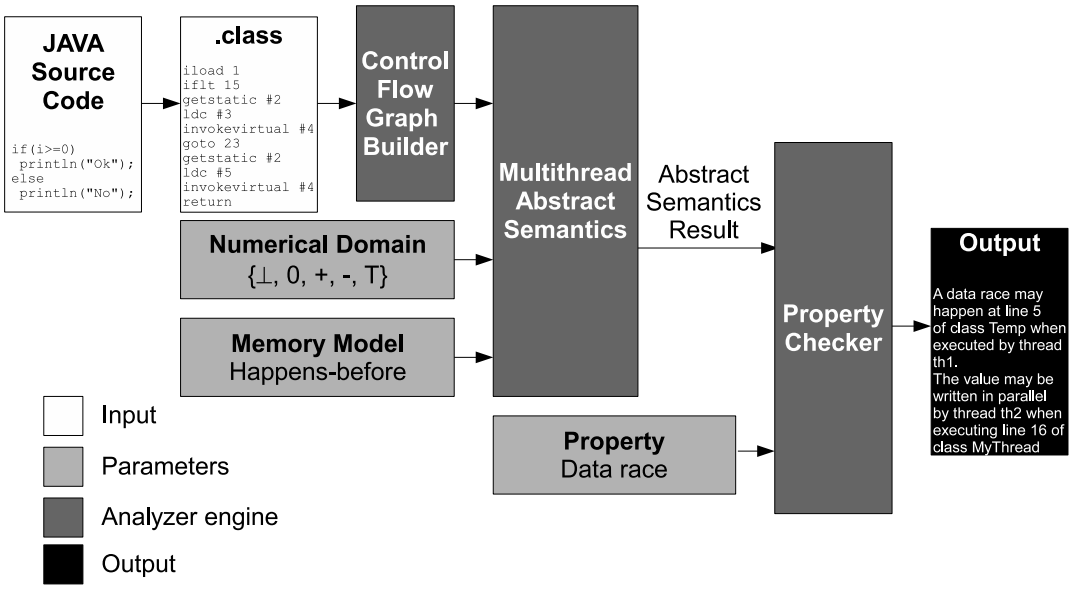

Figure 1. Overall structure of Checkmate

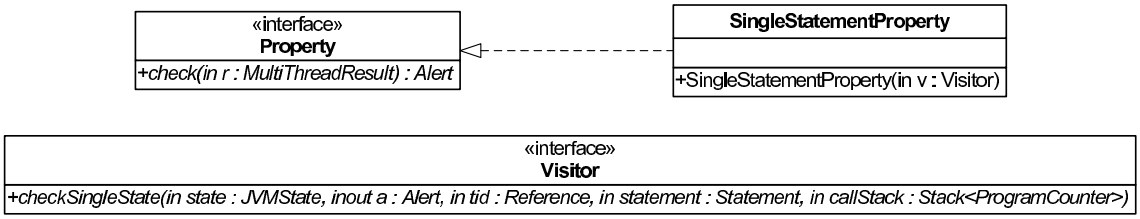

Figure 2. Property interface

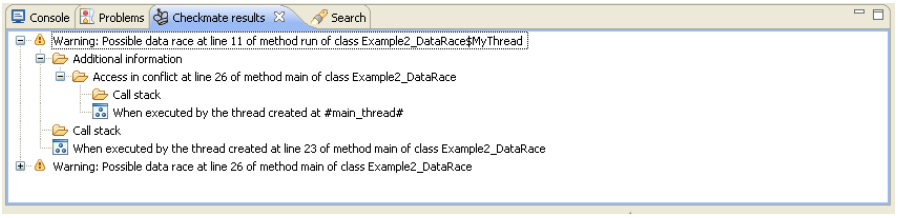

Figure 7. Output

by one file (`commandline/checkmate.jar`). It can be executed launching `java -jar checkmate.jar [...]` where `[...]` contains the parameters of the analysis. In this way the user can set the memory model, the numerical domain, and the property to be used during the analysis. At the end of the analysis the warnings will be displayed on the standard output.

Eclipse **Plugin:** The Eclipse plugin allows to run the analysis in this software development platform. In order to start the analysis, the class to be analyzed has to be opened in the package explorer window. Then the user has to click on "Checkmate". Figure 5 shows it. Once he clicks on it, a dialog will appear (Figure 6). This requires to select which property we want to analyze. Finally, the results of the analysis will be displayed in a view (Figure 7). This view reports the thread that provokes the warning, the call stack followed in order to arrive at that point, and eventually some additional information, e.g. the accesses in conflict if we found a data race. In addition, the user can set which memory model and numerical domain have to be used during the analysis. The default parameters are the happens-before memory model and the Interval domain.

B. *Parameters of* Checkmate

We implemented a representing set of properties, numerical domains, and memory models.

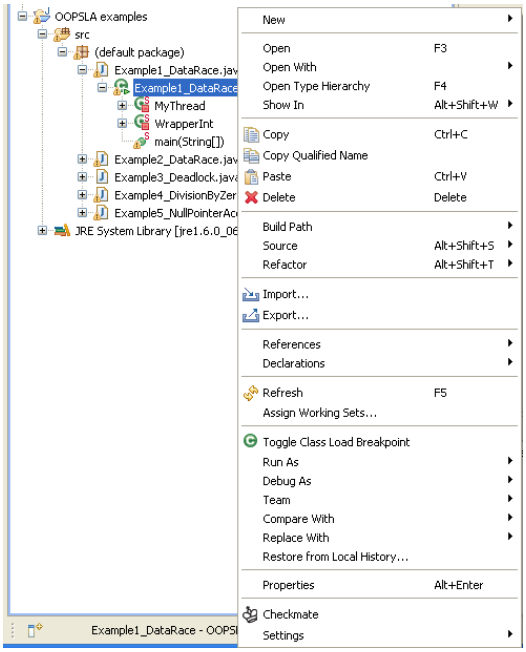

Figure 5. Launching the analysis

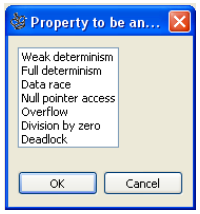

Figure 6. Choosing the property

Properties: Many different properties of multithreaded programs may be interesting. A first set is composed by the ones interesting also at single-thread level, e.g. division by zero. In addition there are other properties specific of concurrent programs, e.g. data race condition. In Checkmate we implemented a representative set of properties of both groups: division by zero, null pointer accesses, overflow, data races, deadlock on monitors, determinism and weak determinism.

Numerical Domains: We implemented some well-known nonrelational abstract domains: Sign [8], Interval [8], Parity [9], and Congruence [23]. These domains are precise enough in order to analyze the properties on which we are interested, but relational domains are required in order to obtain more generic results. In order to apply Checkmate to them, we should first refine our analysis [24], and then we would be in position to apply domains like Polyhedra [25], Octagons [26], Pentagons [4], and Stripes [27].

Memory Models: First of all, we implemented the happens-before memory model. In addition, Checkmate contains two other memory models that are more approximated. The goal of their implementation is to compare the computational overhead induced by more precise memory models. A first abstraction ignores the synchronizations on monitors. The second one abstracts away also the relation that traces when and by whom a thread is launched.

V. Experimental Results

We tested Checkmate on some multithreaded programs presented by [28], [29], [30]. We investigated both the precision and the efficiency of the analysis.
We run it 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.

A. Common Patterns

Lea [29] presented an overview of common patterns when developing concurrent programs in Java. In particular, he introduced some representative examples in order to explain in practice the concepts presented throughout the book. He shows which errors may arise on these examples and how these can be fixed. We apply Checkmate to some examples in order to discover such errors. Usually Lea presents some classes, and then he explains by words why this class has to be considered correct, or which undesirable behaviors may expose. Since Checkmate performs a whole-program analysis, for each example we develop a main method that exposes the behavior of interest.

ExpandableArray: This class implements an array that is automatically expanded if the user wants to append an object when the array is full. All the methods are synchronized. If an user performs in parallel two writes or a read and a write, a conflict arises. In fact, even if all the methods are synchronized, the position of the elements in the array and the read element may be non deterministic because of different interleavings of threads' executions.
This program does not contain data races, and Checkmate precisely discovers it. The conflicts are exposed by the deterministic property, that precisely signals the non deterministic behavior of the two accesses.

LinkedCell: This class implements a list of double values. The methods that read the value contained in the current cell and write a new value are both synchronized. The method that returns the sum of all the cells is not synchronized but it relies on synchronized methods, and thus it does not expose any data race. Finally, a method performs an incorrect sum, reading without synchronized methods the value contained by the first element of the list, thus potentially causing a data race.
Checkmate precisely discovers that the well-synchronized sum method does not expose any data race if executed in parallel with writes on the list. In the same way, it discovers that the ineffectively synchronized sum causes a data race. If we apply the deterministic property we discover that a non deterministic behavior happens even if the program is

well-synchronized.

Document: This class implements a document that contains an enclosure. A synchronized print method that prints the content of the document is provided. Another synchronized printAll method prints all the content using the synchronized print method of the current object, and then invokes the same method on the enclosure. Suppose now to have two documents d1 and d2 whose enclosure is the other document, e.g. the enclosure of document d1 is d2. If we print concurrently these two documents, this may cause a deadlock. For instance the first thread may start the execution of printAll and acquire the monitor of d1. Then the control may switch to the second thread, that acquires the monitor of d2 and yields on the monitor of d1. Finally, the control switches to the first thread, that starts yielding on the monitor of d2, causing a deadlock.

Checkmate soundly discovers that this program may produce a deadlock.

Dot: This class implements a dot in a Cartesian plane. Its coordinates are stored in a Point object. The methods provided by Point class in order to access the information are not synchronized, but all the methods of class Dot are synchronized. On the other hand, if we move a point and shift its x axis value concurrently we may obtain nondeterministic executions.

Checkmate validates this program applying the data race condition, as in fact this program is data race free. In addition, it discovers the nondeterministic behavior applying the deterministic property.

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

TwoLockQueue: This class implements a queue on which we can 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 it. 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. In order to obtain this result, we add an access to a field of the object returned by the take action, and then we analyze this program with the NullPointerException property, that discovers that a NullPointerException may happen. 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 an updatable account, an account holder, and two account recorders, one correct and the other one evil. We refer the interested reader to [29] 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, an evil recorder may cause a nondeterministic behavior.

Checkmate precisely signals it. In particular, if the account recorder is not evil or the account holder use an immutable instance of the recorder, it proves that the program is deterministic. On the other hand, if the account recorder is evil and the account holder does not oblige the use of an immutable instance of the recorder, it signals that a nondeterministic behavior may happen.

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

Table I
THE ANALYZED PROGRAMS

Discussion

Checkmate performs a precise and correct analysis of the representative set of examples we chose from [29]. In particular, in each case it discovers the bug or proves that the program is correct with respect to the behavior of interest. This result is achieved thanks to the high level of flexibility of Checkmate. Using different properties allows us to tune the analysis in order to catch all the bugs. In addition, we found out that the deterministic property is often the only way to discover the behavior of interest. This confirms our impression that this property is in position to break the limits of existing properties applied to multithreaded programs, and in particular of the data race condition.

B. Benchmarks

We apply Checkmate to the analysis of some well-known examples and benchmarks taken from the literature. Two applications (philo, and elevator) are taken from [28], while the others (barrier, forkjoin, sync, sor, crypt, lufact, and montecarlo) are taken from the Java Grande Forum Benchmark Suite [30]. We remove from the original programs only the calls to system functions (e.g. System.out.println) as sometimes they deal with reflection or native methods that are not supported by Checkmate. Table I reports the ana-

Program	Top AP	Top TL	Top HB	Top ST	Sign AP	Sign TL	Sign HB	Sign ST	Int. AP	Int. TL	Int. HB	Int. ST
philo	<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”
barrier	<1”	<1”	<1”	<1”	<1”	1”	1”	<1”	1”	1”	2”	1”
sync	1”	1”	1”	<1”	1”	1”	1”	<1”	2”	2”	3”	1”
crypt	4”	4”	5”	1”	6”	6”	6”	3”	16”	17”	17”	13”
sor	4”	4”	4”	2”	6”	6”	7”	2”	16”	17”	17”	5”
elevator	27”	28”	31”	11”	10”	11”	11”	4”	18”	18”	19”	7”
lufact	25”	25”	27”	10”	52”	52”	53”	20”	5’52”	5’56”	5’59”	2’08”
montecarlo	54”	56”	1’02”	23”	2’23”	2’26”	2’35”	45”	1h00’33”	1h00’38”	1h00’56”	16’48”

Program	Par. AP	Par. TL	Par. HB	Par. ST	Cong. AP	Cong. TL	Cong. HB	Cong. ST	Total AP	Total TL	Total HB	Total ST
philo	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	1”	1”	2”	1”
forkjoin	<1”	<1”	<1”	<1”	<1”	<1”	<1”	<1”	1”	2”	2”	<1”
barrier	<1”	1”	1”	<1”	1”	1”	1”	<1”	3”	4”	4”	2”
sync	1”	1”	1”	<1”	1”	1”	2”	1”	6”	6”	7”	2”
crypt	5”	5”	6”	1”	4”	5”	5”	2”	35”	37”	38”	20”
sor	5”	5”	6”	2”	5”	5”	5”	2”	36”	38”	39”	13”
elevator	29”	30”	30”	11”	28”	29”	29”	11”	1’52”	1’56”	2’00”	46”
lufact	28”	29”	29”	12”	27”	29”	29”	11”	8’04”	8’11”	8’17”	3’02”
montecarlo	1’38”	1’38”	1’43”	31”	54”	1’00”	1’04”	25”	1h06’22”	1h06’39”	1h07’19”	18’52”

Table II
TIMES OF ANALYSIS

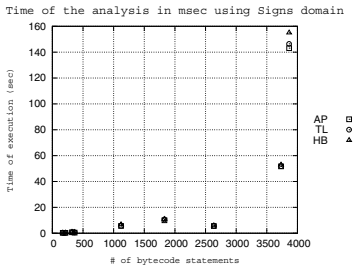

Figure 8. Sign domain

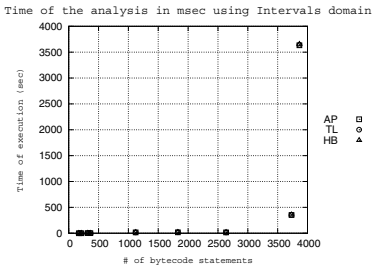

Figure 9. Interval domain

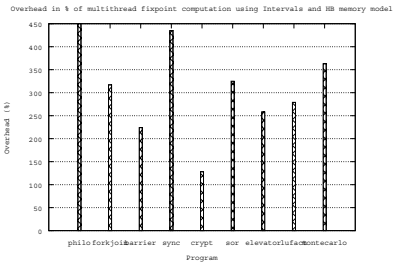

Figure 10. Overhead of fixpoint computation

lyzed programs, the number of statements and the number of abstract threads. Note that in all the cases the abstract threads approximate a potentially unbounded number of concrete threads.

We applied the analysis to all the benchmarks with all the possible combinations of memory models and abstract numerical domains. Table II reports the computational times. For each numerical domain, we report the computational times obtained using the more relaxed memory model (column **AP**), using the memory model that traces only when a thread is launched (column **TL**), using the happens-before memory model (column **HB**), required to compute the semantics of each thread in isolation (column **ST**), i.e. without considering the values written in parallel by other threads.

For programs with less than 500 statements the analysis is quite fast. In addition, the computational times of the same program are comparable using different numerical domains. The analysis of crypt is always quite faster than the one of elevator, even if it is bigger. This happens because of the internal structure of the program.

For Sign and Interval domains, we plot the times of the analysis using the three memory models with respect to the overall number of analyzed statements. Figure 8 depicts the results obtained applying the Sign domain, while Figure 9 the ones obtained with Interval domain. With Sign domain, the time of the analysis does not grow too much with respect to the number of statements. The complexity seems to be almost linear with respect to it.

Intervals do not respect this rule. In particular, the analysis of montecarlo is quite slower. We check if this slowness is due to our overall approach to multithreading that could not scale up, or to the fixpoint computation of a single thread. Figure 10 plots the overhead of the multithread fixpoint computation using Intervals and the happens-before memory model. It turns out that this overhead is always

between 150% and 450%, and so they do not depend on the length of the program. In fact, we often obtain the biggest overhead for the smallest application. In addition, for bigger applications the overhead is almost stable around 300%. This result is quite encouraging: it means that in average we need 3 iterations in order to reach a fixpoint in our multithreaded semantics. In addition, we think that we can improve this result as our implementation is not optimized at all. For instance, we may parallelize the analysis of the single-thread semantics of different threads during the same iteration of the multithreaded semantics.

VI. Related Work

In the literature, some generic analyzers applied to sequential programs, and some analyses of a specific property of concurrent programs have been proposed. In addition, several approaches based on model checking analyze multiple properties on multithreaded programs.

A. Generic Analyzers

Many generic analyzers for object-oriented programming languages have been proposed during the last few years, e.g. Clousot [24], JULIA [31], Cibai [5], and JAIL [32]. Without entering in the details, some of them worked at bytecode level, while others analyze directly the source code. Some of them scale up and achieve better performances w.r.t. Checkmate, but none is sound for multithreaded programs. Thus, as far as we know, Checkmate is the first generic static analyzer of multithreaded programs.

B. Model Checking

Context bound model checking [33] is a novel approach that has obtained a huge amount of both theoretical and practical results during last years. 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 [34], a multithreaded program is analyzed until a given context bound, i.e. the number of context switches is limited to n. Instead, our approach relies on the idea of abstraction. We are able to build up an analysis sound with respect to all the possible multithreaded executions, with an unbounded number of context switches, executing on all the possible multi-core architectures, and considering also a weak memory model. This requires the computation of two nested fixpoint. Intuitively, at the n-th iteration our multithreaded semantics computes all the executions with at most n context switches (and something more, as we consider not only sequentially consistent executions). Iterating this process until a fixpoint is reached accumulates all possible multithreaded executions. Thus we can conclude that our approach is strictly more expressive (but sometimes less precise, as we do not execute the program but we rely on abstraction) than the one proposed by context bound model checking.

Thread-modular model checking [35] tries to reduce the state explosion problem considering each thread separately. The updates that are performed by a thread are summarized through an environment assumption, that is used by other threads to know which values have been written in parallel. Our approach is quite similar to this intuition, as we consider each thread separately when computing its semantics [19]. On the other hand, we can tune our analysis at different levels of precision and efficiency.

Other authors [36] proposed intra-procedural analyses that summarize the concurrent behavior of other procedures. Usually, in order to achieve such level of modularity they consider only well-synchronized programs. Thus they are not sound w.r.t. all the possible multithreaded executions (e.g. programs containing data races).

C. Concurrency Properties

Many approaches were developed in order to statically analyze multithreaded programs. Most of them deals with deadlock and data race detection [37]. First of all, usually these approaches suppose that the execution is sequentially consistent, but this assumption is not legal adopting, for instance, the Java Memory Model. In addition, these analyses are particular for a given property, and they cannot be applied to other properties, while Checkmate has been already applied successfully to a wide set of properties.

Data Race Analysis

The (maybe) most known work of last years has been the type system developed by Abadi, Flanagan and Freund [38]. This work is modular, and it is proved to scale as it was successfully applied to programs of a couple of hundreds of thousands of code lines. On the other hand, it requires manual annotation, and it does not provide any information on possible missing locks when a data race is detected.

Naik and Aiken [39] apply a must not alias analysis through a specific type system in order 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 experimental results seem to underline that the approach can not scale up and they are worse than our results, as this analysis requires more than 3 minutes to analyze only 2 classes.

Kahlon et al. [40] presents a model-checking based analysis in order to statically detect data races. The work has been divided into three different steps: (i) discovering which variables share information (ii) checking through a must alias analysis the owned monitors when shared variables are accessed (iii) reducing the false warnings. The proposed must-alias analysis is quite similar to ours.

Another data race detector based on model checking has been proposed by Henzinger et al. [41]. The analyzed programming language synchronizes through atomic sections, and so it is quite different from the lock-based synchroniza-

tion of Java. Moreover the experimental results are afflicted by the well-known state explosion problem.

Deadlock Detection

Many works are focused on the dynamic detection of deadlocks [42], [43], [44]. These tools are able at runtime to detect if a deadlock happens. On the other hand, if a program may expose a deadlock but we test it only on deadlock-free executions, these tools do not discover the deadlock.

About static analyses, Williams et al. [45] propose an analysis that detects deadlock on synchronized statements and wait invocations. This analysis was implemented and applied to many libraries, and it discovered 14 distinct deadlocks. This analysis makes some assumptions on how an user interacts with the 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 if it calls libraries that do not respect these assumptions.

Awargal et al. [46] present a type system in order to detect at compile time potential deadlocks on synchronized statements. The information inferred by this static analysis is used in order to restrict the runtime checks of possible deadlocks only on locks that are not proved to be deadlock free at compile time. The analysis has not been implemented but only applied by hand. The authors studied in the details the speedup of the runtime that uses this information.

D. Other properties

Many static analysis and tools have been proposed in order to detect accesses to null pointers, divisions by zero, and overflows. Without entering in the details, usually these approaches are sound only for sequential programs and they do not support concurrency. On the other hand, they are often more precise than the ones obtained applying Checkmate to these properties, as we applied our analyzer to this property without developing a specific analysis, e.g. a specific numerical domain.

VII. Conclusion

In this paper we presented Checkmate, a generic static analyzer based on abstract interpretation theory for Java multithreaded programs tuned at bytecode level. It supports all the most relevant features of Java language, as dynamic unbounded threads 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 deeply the experimental results obtained when applying Checkmate to some common patterns of concurrent programming in Java [29], and to a set of well-known benchmarks [28], [30]. The precision exposed when analyzing these examples is quite encouraging, and we are in position to analyze programs with an unbounded number of threads and thousands of statements in a limited time.

A. Future Work

Future work concerns the refinement of the analysis and its application to industrial software. In particular, we want to refine the abstract domain in order to apply relational domains. In order to reach this goal, we need to perform some transformations on the bytecode, e.g. stack abstraction and expression recovery [24]. Our aim is also to refine our memory model in order to trace more synchronization actions, and with some restrictions of the Java memory model that are disregarded in the happens-before one.

Acknowledgment

This research is partially supported by "MIUR PRIN'07 Project SOFT - Tecniche formali orientate alla sicurezza".

References

[1] E. A. Lee, "The problem with threads," in *Computer*. IEEE Computer Society Press, 2006.

[2] G. Koch, "Discovering multi-core: extending the benefits of Moore's law," in *Technology Intel Magazine*. Intel, July 2005.

[3] H. Sutter and J. Larus, "Software and the concurrency revolution," in *ACM Queue*. ACM Press, 2005.

[4] F. Logozzo and M. Fähndrich, "Pentagons: A weakly relational domain for the efficient validation of array accesses," in *SAC '08*. ACM Press, 2008.

[5] F. Logozzo, "Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes," in *VMCAI '07*, ser. LNCS. Springer-Verlag, 2007.

[6] F. Spoto, "The Julia Generic Static Analyser," `http://profs.sci.univr.it/~spoto/julia/`.

[7] I. Pollet and B. Le Charlier, "Towards a complete static analyser for java: an abstract interpretation framework and its implementation," in *AIOOL '05*, ser. ENTCS. Elsevier, 2005.

[8] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77*. ACM Press, 1977.

[9] ——, "Systematic design of program analysis frameworks," in *POPL '79*. ACM Press, 1979.

[10] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] M. Odersky, *The Scala Language Specification*, 2008.

[12] B. Meyer, *Object-Oriented Software Construction (2nd Edition)*, ser. Professional Technical Reference. Prentice Hall, 1997.

[13] P. Ferrara, "Static analysis via abstract interpretation of multithreaded programs," Ph.D. dissertation, Ecole Polytechnique of Paris (France) and University "Ca' Foscari" of Venice (Italy), May 2009.

[14] P. Cousot, "The calculational design of a generic abstract interpreter," in *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[15] W. Pugh, "The Java memory model is fatally flawed," in *Concurrency - Practice and Experience 12(6)*. Wiley, 2000.

[16] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model." in *POPL '05*. ACM Press, 2005.

[17] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs." in *IEEE Trans. Computers*, 1979.

[18] ——, "Time, clocks, and the ordering of events in a distributed system," in *Commun. ACM*. ACM Press, 1978.

[19] P. Ferrara, "Static analysis via abstract interpretation of the happens-before memory model," in *TAP '08*, ser. LNCS. Springer-Verlag, 2008.

[20] ——, "A fast and precise analysis for data race detection," in *Bytecode '08*, ser. ENTCS. Elsevier, 2008.

[21] ——, "Static analysis of the determinism of multithreaded programs," in *SEFM '08*. IEEE Computer Society, 2008.

[22] A. S. Foundation, "Bcel - the bytecode engineering library," 2006. [Online]. Available: jakarta.apache.org/bcel/

[23] P. Granger, "Static analysis of linear congruence equalities among variables of a program," in *Proceedings TAPSOFT '91*, ser. LNCS. Springer-Verlag, 1991.

[24] F. Logozzo and M. Fähndrich, "On the relative completeness of bytecode analysis versus source code analysis," in *CC '08*, ser. LNCS. Springer-Verlag, 2008.

[25] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL '78*. ACM Press, 1978.

[26] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, 2006.

[27] P. Ferrara, F. Logozzo, and M. Fähndrich, "Safer unsafe code for .net," in *OOPSLA '08*. ACM Press, 2008.

[28] C. Von Praun and T. R. Gross, "Object race detection," in *OOPSLA '01*. ACM Press, 2001.

[29] D. Lea, *Concurrent Programming in Java*. Addison-Wesley, 1996.

[30] Java Grande Forum Benchmark Suite, `http://www.epcc.ed.ac.uk/research/activities/java-grande/`.

[31] F. Spoto, "JULIA: A Generic Static Analyser for the Java Bytecode," in *FTfJP'2005*, 2005.

[32] P. Ferrara, "JAIL: Firewall analysis of java card by abstract interpretation," in *EAAI '06*, 2006.

[33] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *TACAS '05*, ser. LNCS. Springer-Verlag, 2005.

[34] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 416–430, 2000.

[35] C. Flanagan and S. Qadeer, "Thread-modular model checking," in *SPIN '03*. Springer, 2003.

[36] S. Qadeer, S. K. Rajamani, and J. Rehof, "Summarizing procedures in concurrent programs," in *POPL '04*. ACM, 2004.

[37] M. C. Rinard, "Analysis of multithreaded programs," in *SAS '01*, ser. LNCS. Springer-Verlag, 2001.

[38] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," in *TOPLAS '06*. ACM Press, 2006.

[39] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," in *POPL '07*. ACM Press, 2007.

[40] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *CAV '07*, ser. LNCS. Springer-Verlag, 2007.

[41] T. A. Henzinger, R. Jhala, and R. Majumdar, "Race checking by context inference," in *PLSI '04*, 2004.

[42] S. Bensalem, J. Fernandez, K. Havelund, and L. Mounier, "Confirmation of deadlock potentials detected by runtime analysis," in *PADTAD '06*. ACM Press, 2006.

[43] S. Bensalem and K. Havelund, "Scalable dynamic deadlock analysis of multithreaded programs," in *PADTAD '05*. ACM Press, 2005.

[44] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, 2007.

[45] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *ECOOP '05*, ser. LNCS. Springer-Verlag, 2005.

[46] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and runtime monitoring," in *PADTAD '05*. Springer-Verlag, 2005.