

No Java without Caffeine

A Tool for Dynamic Analysis of Java Programs

Yann-Gaël Guéhéneuc*, Rémi Douence†, Narendra Jussien

École des Mines de Nantes
4, rue Alfred Kastler – BP 20823
44307 Nantes Cedex 3
France

E-mail: {guehene|douence|jussien}@emn.fr

Abstract

To understand the behavior of a program, a maintainer reads some code, asks a question about this code, conjectures an answer, and searches the code and the documentation for confirmation of her conjecture. However, the confirmation of the conjecture can be error-prone and time-consuming because the maintainer has only static information at her disposal. She would benefit from dynamic information. In this paper, we present CAFFEINE, an assistant that helps the maintainer in checking her conjecture about the behavior of a Java program. Our assistant is a dynamic analysis tool that uses the Java platform debug architecture to generate a trace, i.e., an execution history, and a Prolog engine to perform queries over the trace. We present a usage scenario based on the *n*-queens problem, and two real-life examples based on the Singleton design pattern and on the composition relationship.

1. Motivation

Soloway *et al.* [27] characterize software understanding activities as composed of inquiry episodes, during which a maintainer reads some code, asks a question about this code, conjectures an answer, and searches the code and the documentation for confirmation of the conjecture.

However, as mentioned in [21], the “*Search in software understanding is very error-prone; a maintainer does not always know where to look for information*”

to support their conjectures.” Our aim is to develop an assistant that helps the maintainer who possesses a limited structural knowledge of an object-oriented program to understand its behavior [4, 29]. We develop CAFFEINE [16], a tool that helps a maintainer to check her conjectures about Java programs, and to understand the correspondence between the static code of the programs and their behavior at runtime.

CAFFEINE generates and analyzes on the fly the trace of a Java program execution, according to a query written in Prolog. The trace is composed of execution events. Events relate to the language constructs. They have semantics reflecting both the control flow and the data flow of the program.

We now present a simple scenario based on an algorithm to solve the *n*-queens problem. This scenario emphasizes the methodology a maintainer follows to express and to verify her conjectures about a program behavior. For the sake of clarity, we choose a simple program and we focus on the methodology rather than on the scenario; Section 4 presents two real-world examples of dynamic analyses.

The *n*-queens problem

Given a $n \times n$ -size chessboard and n queens, how to place the n queens so that they do not attack one another?

The maintainer inherited a Java program implementing an algorithm that solves this problem. After having read the code and understood the overall behavior of the algorithm, the maintainer focuses on the main loop of the algorithm:

*This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

†This work is partly funded by the European project “EasyComp” (www.easycomp.org), no. IST-1999-014191

Main loop of the algorithm

The `placeNextQueen(int column)` method implements the algorithm solving the problem:

```
1 boolean placeNextQueen(final int column) {
    if (column == this.n) {
        this.print();
        return true;
5    }
    else {
        int row = 0;
        while (row < this.n) {
            if (!this.attack(row, column)) {
10                this.setQueen(row, column);
                if (this.placeNextQueen(column + 1)) {
                    return true;
                }
            }
            else {
15                this.removeQueen(row, column);
            }
            row++;
        }
20        return false;
    }
23 }
```

From the main loop, the maintainer understands the algorithm strategy:

Algorithm strategy

The algorithm uses a trial-and-error strategy, implemented with a backtrack mechanism. The algorithm works in three steps:

1. The algorithm checks if placing a queen at `row` and `column` is legal (method `attack(int row, int column)`, line 9, starting with `row 0` and `column 0`), *i.e.*, if at this position, the queen does not attack any other already-present queen.
2. If the position is legal, the algorithm places the queen (method `setQueen(int row, int column)`, line 10) and it goes on with the next column (method `placeNextQueen(int column)`, line 11).
3. If the algorithm cannot place legally the next queen, it backtracks by removing the previously-placed queen (method `removeQueen(int row, int column)`, line 15) and it tries again in the same column with the next row (statement `row++`, line 18).

The maintainer feels that this strategy involves numerous backtracks. She analyzes the execution trace to get the number of backtracks.

Counting the number of backtracks

The number of backtracks corresponds to the number of calls to the `removeQueen(int row, int column)` method, this translates into the following query:

```
1 query(N, M) :-
    nextMethodEntryEvent(removeQueen),
    N1 is N + 1,
    query(N1, M).
5 query(N, N).
```

The maintainer instructs CAFFEINE to increment recursively a counter every time a call to the `removeQueen()` method occurs in the trace. The analysis answers that there are 105 backtracks. Now,

she conjectures that the algorithm removes frequently (*w.r.t.* the total number of backtracks) a queen just after having placed her: She conjectures that there are several immediate backtracks. She turns to CAFFEINE for an answer.

Counting the number of immediate backtracks

The number of immediate backtracks corresponds to the number of times the algorithm control flow matches the pattern of successive method invocations:

`setQueen() ↦ placeNextQueen() ↦ removeQueen()`

The verification of the pattern translates into the query:

```
1 query(N, M) :-
    nextMethodEntryEvent(setQueen),
    nextPlaceNextQueen,
    nextMethodEntryEvent(removeQueen),
5    % Count and repeat the query (details omitted).

nextPlaceNextQueen :-
    nextMethodEntryEvent(NAME),
    isPlaceNextQueen(NAME).
10
isPlaceNextQueen(placeNextQueen) :- !.
isPlaceNextQueen(removeQueen) :- !, fail.
isPlaceNextQueen(setQueen) :- !, fail.
14 isPlaceNextQueen(_) :- nextPlaceNextQueen.
```

The maintainer instructs CAFFEINE to reach the point where the control flow enters method `setQueen()` using the predicate `nextMethodEntryEvent(setQueen)`, line 2. From that point, she commands CAFFEINE to reach the point where the control flow enters method `placeNextQueen()`, line 3, using the predicate `nextMethodEntryEvent(NAME)`; `NAME` must correspond to method `placeNextQueen()`, line 11: If `NAME` corresponds to method `removeQueen()` or `setQueen()`, lines 12 and 13, the current control flow does not match the expected pattern and the search resumes¹; If `NAME` corresponds to any other method, line 14, the search goes on. Finally, she directs CAFFEINE to reach the point where the control flow enters method `removeQueen()`, line 4.

The answer to the maintainer's question, with the previous `n`-queens algorithm is 42: The algorithm removes 42 times a queen just after having placed her. The maintainer obtains a better understanding of the algorithm behavior, using dynamic analysis: Now, she could suppress the immediate backtracks by using a look-ahead algorithm [20].

Such a scenario highlights the features of our tool. We now detail its trace model and its execution model (Section 2). Then, we present the implementation of our tool: Its architecture, some interesting technical issues, and performance measurements (Section 3). We

¹The `!, fail` clause trims the derivation tree of the `isPlaceNextQueen` goal (thus preventing backtrack on this goal) and forces the backtrack on the `nextPlaceNextQueen` goal.

also present two real-life examples and briefly discuss the opportunity of reasoning on a finite set of execution traces (Section 4). Finally, we conclude by presenting related work (Section 5) and future work (Section 6).

2. An execution model for Java

We now present in details the trace model, which describes the model CAFFEINE possesses of the history of execution events. Then, we detail the execution model, which characterizes the content of the trace model, *i.e.*, what are the events generated during analysis.

Trace Model In CAFFEINE, we model the execution as a trace, which is a history of execution events. We do not consider *post-mortem* trace: We do not assume that the complete history is fully available. We can request the next available execution events, but we cannot request previous ones. It is possible, however, to store (part of) the past trace².

We request the next available execution events using Prolog. Prolog runs as a co-routine of the program being analyzed. It executes a query that contains predicates to drive the program execution and to obtain next events. We use Prolog because it possesses a mechanism of unification and high-level pattern-matching capabilities, and because it is expressive, through its backtrack mechanism. Prolog already showed its adequacy to query traces in different works [10, 11]. However, we only use Prolog to demonstrate the feasibility and the usefulness of our approach, we could use other paradigms to express queries, such as regular expressions (for their simplicity and efficiency but with less power of expression) or temporal logic [19].

The predicate to control the execution of the program being analyzed is `nextEvent/3`:

```
nextEvent([<list of class filters>,
          [<list of desired events from the program>], E)
```

This predicate monitors the classes (and their instances) specified by the class filters and runs the program until the next desired event occur in the monitored classes and instances. The event is unified with the variable `E`.

Execution Model Our model of a Java program execution consists in a trace of execution events. We focus on the object model of Java, therefore our events relate to: Classes (load and unload); Fields (access and modification); Constructors, finalizers, and methods (entry and exit). We describe these events as Prolog terms, in the form of a functor and a set of parameters. Table 1 presents the list of all the possible

²A tool exists also for the analysis of *post-mortem* traces generated by CAFFEINE.

events generated by a program execution, available to the maintainer to describe her conjectures.

Finer-grain conjectures require finer-grain events from the method body (declaration, access, and modification of local variables) and from the control structures (`if`, `while`, `for`). We shall consider finer-grain event in next releases of CAFFEINE.

3. Implementation

We now detail the architecture of CAFFEINE and some implementation issues. We want our tool to be portable across platforms and J2SDK versions, therefore we build it as a 100%-pure Java program. We do not use a modified Java Virtual Machine (JVM), possibly losing in performances, and which results in challenging implementation issues, as we show in this section.

Architecture The execution of a program written with the Java programming language takes place in a JVM, which in turn runs on the operating system of a computer. This layered architecture eases the debugging of Java programs. Indeed, the JVM provides a standard interface to debug the program that it runs, the Java Platform Debug Architecture (JPDA) [28].

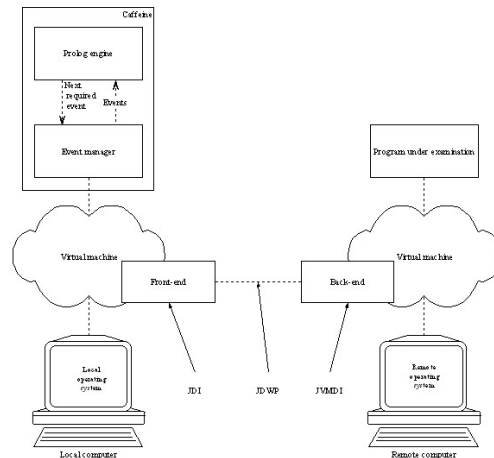


Figure 1. The JPDA

As shown on Figure 1, the JPDA decomposes into three main parts. On the local computer, the local JVM provides a front-end for remote debugging, the Java Debug Interface (JDI). The JDI is a 100%-pure Java interface for debugging a Java program running on a remote JVM. It allows the connection with an already-existing remote JVM or the creation of a remote JVM, and interacting with it. The JDI dialogs with the remote JVM running on the remote computer through the specialized Java native interface for

Java execution event	Definition and parameters
fieldAccess(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>)	The instance field <Field name> of the instance identified by its ID <Unique ID of the instance possessing this field> shall be read.
fieldModification(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>, <Unique ID of the new object assigned to this field>, <Fully qualified name of the class of the new object>)	The instance field <Field name> of the instance identified by its ID <Unique ID of the instance possessing this field> shall be modified. The ID of the object to be assigned to the field is <Unique ID of the new object assigned to this field>. For visualization purpose, the fully-qualified name of the object class is given.
classLoad(<Class name>, <Event unique ID>)	The program requires the class <Class name>.
constructorEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the newly created instance>)	A new instance of class <Declaring class name> is being created. This instance is uniquely identified by <Unique ID of the newly created instance> for the rest of the program execution.
finalizeEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the being-finalized instance>)	The instance of class <Declaring class name> uniquely identified by <Unique ID of the being-finalized instance> is being finalized.
methodEntry(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the class declaring this method>)	The method <Method name> of class <Fully-qualified name of the class declaring this method> is called on the instance uniquely identified by <Unique ID of the receiver instance>.
Identical definitions for dual events: classUnload; constructorExit; and, finalizerExit. Definition changes for event methodExit:	
methodExit(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the class declaring this method>, <The value returned by this method>)	The method <Method name> of class <Fully-qualified name of the class declaring this method> is completing its call on the instance uniquely identified by <Unique ID of the receiver instance>. The returned value is provided in <The value returned by this method>.

Table 1. List of the events from the program execution, available to the Prolog engine.

third-party debugging tools, the JVM Debug Interface (JVMDI). The Java Debug Wire Protocol (JDWP) abstracts the communication layer between the local JVM and the remote JVM.

In the local JVM, The JDI represents the remote JVM as an instance implementing the `com.sun.jdi.VirtualMachine` interface. Through the `VirtualMachine` interface, the local program receives events and has access to instances, classes, and threads of the remote program. When the remote program emits events, the remote JVM suspends its activity (all its threads) and waits for the local program to resume its execution, through a call to the appropriate method of the `VirtualMachine` interface.

In CAFFEINE, we use the JPDA because it offers a powerful API for program-execution instrumentation and because it is portable across platforms and J2SDK versions. We use the JDI to create a remote JVM, to run on this remote JVM the program to analyze, and to obtain events related to the execution of the program being analyzed.

CAFFEINE, Figure 1, decomposes in two main parts: The `EventManager` class and the Prolog engine. The `EventManager` class uses the JDI to control the remote JVM: It indicates to the remote JVM the expected events; It collects the events from the remote JVM; It translates the events into Prolog-compliant forms; It resumes the execution of the remote JVM as requested

by the Prolog engine.

The Prolog engine solves the query that describes the analysis to perform; query expressed in terms of trace model and Java execution model. We implement the Prolog engine using JIProlog [6]. JIProlog is a Prolog engine implemented in Java, with advanced extension capabilities. In particular, it is possible to write a custom predicate that, when evaluated by Prolog, calls some Java code with a Prolog term as input and unifies its result with a Prolog variable as output. We implement the `nextEvent/3` predicate as a custom predicate. This custom predicate creates new event requests, sets the appropriate filters, and resumes the execution of the remote program to obtain the next desired event from the remote JVM, through the `EventManager` class.

A specific final class, the `Caffeine` class, offers a unique static method `void run(...)`. This method creates a remote JVM and initializes it with the appropriate arguments. Then, it creates a new Prolog engine with the user query and starts the resolution of the query.

The `NQueensCaffeineLauncher` class contains the Java code that launches the analysis of the Java implementation of the n-queens algorithm:

```

1 public final class NQueensCaffeineLauncher {
    public static void main(final String[] args) {
        Caffeine.run(
            "Caffeine/Example/Queens/Rules.pl",
            <Classpath omitted here>,
5

```

```

    "caffeine.example.queens.NQueens",
    new String[] { "caffeine.example.queens.*" },
    Caffeine.GenerateMethodEntryEvent,
    null);
10     }
11 }

```

It consists in a call to the `run(...)` method of the `Caffeine` class, with the appropriate arguments:

- Line 4: The name of the file that contains the Prolog query to verify.
- Line 5: The program execution *classpath*.
- Line 6: The main class of the program to analyze.
- Line 7: The event filters.
- Line 8: The event required for the analysis. In our example, the analysis only requires the JVM to generate method-entry events.
- Line 9: A list of fields to monitor, the `null` value tells `CAFFEINE` that no field needs monitoring.

The following is the output generated when running the `NQueensCaffeineLauncher` class, *i.e.*, the output resulting from the analysis of the n-queens algorithm, answering the maintainer's conjecture about the number of immediate backtracks:

```

JIProlog v1.9.1.2
By Ugo Chirico - http://www.ugosweb.com/jiprolog
<Details of the output omitted>
I removed a queen after having placed her for the 42 time(s)
(Remote JVM) 0 4 7 5 2 6 1 3
Solution = main(0, 42)
JVM running time: 138890 ms.
(37 ms. per step for the 3662 steps.)

```

The solution appears in the last output line of the Prolog query, `I removed a queen after having placed her for the 42 time(s)`, and in the result of the query, `Solution = main(0, 42)`.

We now point at some technical issues met with the JPDA, which prevented the generation of interesting events, and our novel solutions.

Returned value By definition, the method-exit event generated by the remote JVM does not provide the value being returned by non-void methods. We overcome this limitation by wrapping any returned value in a special method, at load-time and at the byte-codes level, using `CFPARSE` [15]. The special method accepts one parameter and immediately returns it. The `EventManager` deals with a method exit event from the special method by storing the value of its parameter, which corresponds to the returned value.

Let consider the code of the previous `boolean placeNextQueen(final int column)` method:

```

1 boolean placeNextQueen(final int column) {
  <Details omitted>
  return true;
  <Details omitted>
5  return true;
  <Details omitted>
  return false;
8 }

```

We wrap the returned values by the method before returning them. The following code pictures the method new implementation³:

```

1 boolean placeNextQueen(final int column) {
  <Details omitted>
  return CaffeineWrapper.methodReturnedValue(true);
  <Details omitted>
5  return CaffeineWrapper.methodReturnedValue(true);
  <Details omitted>
  return CaffeineWrapper.methodReturnedValue(false);
8 }

```

The wrapper method directly forwards its parameter as returned value:

```

1 public abstract class CaffeineWrapper {
  public static final boolean methodReturnedValueWrapper(
    final boolean value) {
    return value;
5  }
  <Other methods omitted.>
7 }

```

The introduction of a method call for each return is a solution to obtain the returned value. Another solution could be to monitor the stack frame and to record the value on top of the stack before the JVM pops the corresponding frame.

Finalizers The garbage-collector calls the `void finalize()` method on an instance when it determines that there are no more reference to this instance. However, for speed optimization and safety reasons, a JVM hardly calls finalizers. We use specifically-tailored 100%-pure-Java strategies to ensure the calls to finalizers.

First, there is, in general, no insurance that the JVM calls a finalizer when an instance is ready for garbage-collection. We force the JVM to call the finalizers on exit, using the `void runFinalizersOnExit(boolean)` method implemented by the `System` class, although this method is deprecated. Also, we add a thread that periodically calls the garbage-collector and the finalization mechanism, to stimulate calls to the finalizers before the remote JVM exits.

Second, for speed optimization purpose, the JVM does not call the `void finalize()` method defined in the `Object` class: A class must explicitly override the finalizer for the JVM to call this method. We use our own class-loader in the remote JVM to add, when

³For the sake of clarity, we present the transformations at the source-code level.

required, a finalizer to loaded classes, using JAVASIST [5]. The added finalizer delegates its operation to its `super` finalizer.

However, these strategies do not fully satisfy our expectations regarding finalizers. We plan to study the feasibility of implementing a dedicated garbage-collector that calls the finalizer method as soon as no more reference points to an instance, and that ensures, if possible, a partial order among the calls to finalizers when the JVM exits. Reference counting is a candidate technique for such a garbage-collector, we could use the Java Native Interface for its implementation.

Program end An interesting property of the Java execution model is that each program possesses a unique entry point: A public class with a main method of signature `public static void main(String[])`⁴. Thus, it is simple to generate an event associated with the beginning of a program.

On the other hand, a Java program may have several exit points. The program-end corresponds to the termination of the `main(String[])` method or to the termination of the `run()` methods of all running threads (normal return or exceptional return), or results from a call to the `System.exit(int)` method.

These different exit points do not have equivalent effects on the JVM. We now compare a program-end resulting from the termination of all the running threads (normal termination or exceptional termination) with a program-end resulting from a call to the `System.exit(int)` method.

A simple multi-threaded Java program

The following code represents a simplistic multi-threaded Java program. The `main(String[])` method implicitly executes in a `main` thread; Within this method, a new thread starts that sleeps for 2 seconds and then returns:

```
public class MultiThreading {
    public static void main(final String[] args) {
        new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.currentThread().sleep(2000);
                }
                catch (final InterruptedException ie) {
                    ie.printStackTrace();
                }
                System.out.println(
                    Thread.currentThread().getName() +
                    " just terminated.");
                // System.exit(0);
            }
        }, "son").start();
        System.out.println(
            Thread.currentThread().getName() +
            " just terminated.");
    }
}
```

⁴We consider that two different public classes, each with a main method, represent two different programs.

Table 2 presents subsets of the traces obtained with the execution of the `MultiThreading` program, first with all threads returning, and then with the second thread invoking the `System.exit(int)` method. The two traces are different:

- When all the threads return, the JVM generate events related to the thread terminations ((`main just ended.`) and (`son just ended.`)).
- When a thread performs a `System.exit(int)` method invocation, the JVM does not generate any event related to the termination of the current thread or of any other still-running threads. (Moreover, the JVM starts some private threads for its own purposes, such as `Thread-0`.)

These traces suggest the strategy to detect program termination: We must trace calls to the `System.exit(int)` method, and we must trace the termination of user-defined threads.

We trace calls to the `System.exit(int)` method by replacing them, using `CFPARSE`, with calls to the unique `CaffeineWrapper.caffeineUniqueExit(int)` method, which we can trace efficiently. We trace the termination of user-defined threads using the JVM-generated events `ThreadStartEvent` and `ThreadDeathEvent`.

Termination of the MultiThreading program
<pre><Other events omitted> (main just ended.) (Thread-1 just started.) (Remote JVM) son just terminated. methodExit(8, run, MultiThreading\$1, void) (son just ended.) finalizerEntry(9, MultiThreading\$1) finalizerExit(10, MultiThreading\$1) classUnload(11, MultiThreading) classUnload(12, MultiThreading\$1)</pre>
<pre><Other events omitted> (main just ended.) (Thread-1 just started.) (Remote JVM) son just terminated. (Thread-0 just started.) (Thread-0 just ended.) finalizerEntry(8, MultiThreading\$1) finalizerExit(9, MultiThreading\$1)</pre>

Table 2. Events obtained with the termination of the MultiThreading program.

First row, all threads (normally or exceptionally) return; Second row, a thread invokes the `System.exit(int)` method.

Unique identifiers We want any instance created in a JVM to possess a unique identifier. We could access a unique identifier of an instance either through the `JDI`, or using the `int hashCode()` method defined in the `Object` class, or using the `int identityHashCode(Object)` method of class `System`. However, we cannot use these strategies within `CAF-FEINE`.

First, the JDI associates a unique identifier with each instance in the remote JVM. However, the scope of this unique identifier is limited to a single thread: If an instance is referenced to by two different threads, it has two different unique identifiers, *w.r.t.* the considered thread. Generation of finalizer-related events relies on multi-threading, hence, we cannot use the unique identifier provided by the JDI.

Second, we cannot rely on the `int hashCode()` method because a particular class may override it and make it unusable: Let assume we request the unique identifier of an instance for which the control flow just entered a constructor, this instance being not yet initialized, the overridden `int hashCode()` method may throw an `Exception`.

Third, we cannot call the `int identityHashCode(Object)` method on an instance existing in a remote JVM because of a limitation among the mechanisms of remote method invocation provided by the JDI and the garbage-collection mechanisms of the remote JVM. This limitation⁵ prevents the garbage-collector to collect an instance after a remote method invocation, and thus prevents the finalizer to run and the corresponding event to happen.

Consequently, in CAFFEINE, we add to each class, at load-time, an instance field that holds a unique identity number by modifying the inheritance graph with JAVASSIST. We set the unique identifier number on instantiation.

Performances Table 3 shows performance results for the n-queens implementation in Java and for the `MoneyTest` class, from JUNIT v3.7. We measure those results on a Pentium-III processor, with 256 Mb of RAM, Windows 2000, running CAFFEINE and the examples with the J2SDK v1.4.0 in the ECLIPSE [25] development environment. We give the execution-time as the average of 30 successive runs.

These measures show that the JVM event generation mechanisms dramatically slow down the program execution. Figure 2 illustrates that execution-time may increase exponentially with the number of events generated by the JVM. The execution-time increase does not hinder the use of CAFFEINE on real-life programs (see Section 4), however it implies to consider carefully the required events and their associated filters.

Future work includes investigating the use of reflective systems or aspect-orientation techniques to generate events more efficiently.

⁵The authors noticed this limitation for the JPDA implementation of the J2SDK v1.3.1 and of the J2SDK v1.4.0.

	N-Queens algorithm		MoneyTest	
	Time (in sec.)	Increase Factor	Time (in sec.)	Increase Factor
Stand-alone	0.020	/	0.194	/
Debug mode				
Without events	0.040	2.000	0.238	1.227
With events	109.347	5467.350	14.935	76.988
CAFFEINE (No analysis)				
Without events	0.270	13.500	0.818	4.128
With events	112.943	5647.150	32.700	168.557
CAFFEINE (With analysis)	116.149	5807.450	86.928	448.082

Table 3. Performances and relative time increase-factor for two different Java programs and analyses.

N-Queens algorithm: There are 1209 method-entry events for the main algorithm class.

MoneyTest class: There are 222 constructor-entry, field-modification, finalizer-exit, and program-end events for the `MoneyTest` and `MoneyBag` classes.

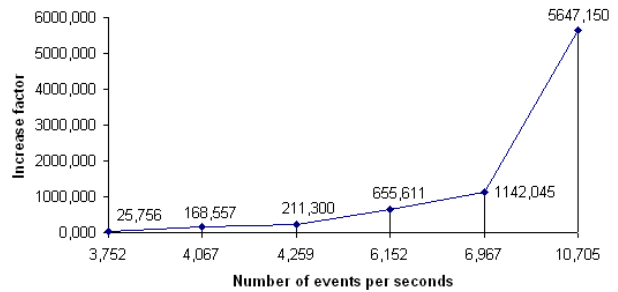


Figure 2. Increase factor of execution time w.r.t. the number of method-entry events generated by the JVM per second.

Figures given for different test-programs and filters: SNAKE, `MoneyTest`, two different queries on JAVAHA NOI, CACAO, and the Java implementation of the n-queens algorithm [18].

Increase factor = Caffeine (no analysis) execution-time / Stand-alone execution-time

Number of events per seconds = Number of events generated by the JVM / Stand-alone execution-time.

4. Using Caffeine in real-life

We now sketch two real-life examples of conjectures about different Java programs and the solutions CAFFEINE offers to verify them. Because of the lack of space, we only present the problems and their solutions. The first conjecture concerns the implementation of the Singleton design pattern in JHOTDRAW v5.2. The second conjecture concerns the relationships among classes in the JUNIT v3.7 framework.

Singleton design pattern The JHOTDRAW v5.2 [12] framework is a graphic editor offering different geometrical forms, colors, fonts, and animations. The framework is easily extensible through the use of well-documented extension points, and of several design patterns [14].

A maintainer looks at the framework architecture and browses the `ch.ifa.draw.util` package. This package contains a class `IconKit`, which documentation states as being a singleton, but which constructor is public (allowing more than one instance to be built). She wonders if this class is really a singleton.

To verify her conjecture about the `IconKit` class being a singleton, she instructs `CAFFEINE` to count the number of times the `IconKit` class is instantiated [18], using a fairly simple query (12 clauses).

After running a typical session with `JHOT-DRAW v5.2`, she finds out that the `IconKit` class is instantiated once and only once and therefore is a singleton. She modifies the `IconKit` implementation to reflect strictly the Singleton design pattern and to prevent future undue instantiation.

Binary Class Relationships A maintainer wants to represent the `JUNIT v3.7` framework [13], using the UML visual modeling language. In the UML, there exist three different binary class relationships [17]:

- Association: An association between two classes **A** and **B** is the ability of an instance of **A** to send a message to an instance of **B**, with the possibility of mutual associations between the instances.
- Aggregation: An aggregation relationship exists when the definition of one class contains instances of the other class. It distinguishes a *whole* from a *part*. The aggregate class must define a field of the type of the aggregated class (or an array field, or a collection). Instances of the aggregate class send messages to the referenced instance of the aggregated class.
- Composition: A composition is an aggregation between two classes, with a constraint between the lifetime of the instance of the *whole* and the lifetime of the instances of its *part*; and a constraint on the ownership of the instances of the *part* by the instance of the *whole*. The instances of the *part* must not belong to any other *whole*: They are exclusive to the instance of the *whole*.

The maintainer conjectures that composition relationships exist between classes `TestResult` and `TestFailure`, and classes `TestSuite` and `Test`, from package `junit.framework`.

It is difficult to verify statically these relationships because they involve lifetime and exclusivity properties of the instances of the classes. The maintainer writes a query that analyzes the lifetime and exclusivity properties of classes. For lack of space, we do not provide here the query, which is relatively complex (41 clauses).

The maintainer analyzes the `JUNIT v3.7` framework with `CAFFEINE` by running a modified version of the `junit.sample.money.MoneyTest` class, which uses the Swing-based UI and in which some tests fail. The modified version of the `MoneyTest` class covers a greater part of the framework code.

The maintainer finds that the `TestFailure` class is in composition relationship with the `TestResult` class, while the `Test` class is *not* in composition relationship with the `TestSuite` class. Indeed, the exclusivity property between `TestSuite` and `Test` do not hold: In case of failures and errors, the `JUnit` framework shares the instances representing the faulty tests, stored in an instance of class `TestSuite`, with an instance of the `junit.swingui.TestTreeModel` class.

Therefore, the relationships between classes `TestResult` and `TestFailure`, and classes `TestSuite` and `Test` shall be represented in the UML as an aggregation relationship. A discussion between the maintainer and the authors of the framework could start to state whether these are the desired relationships or the results of a design fault.

Discussion The previous example highlights the main drawback of dynamic analysis: The maintainer verifies her conjecture only on a subset of all the possible execution-traces of a program; She might be misled into an erroneous general conclusion.

This drawback does not invalidate the interest of dynamic analysis. Maintainers must use dynamic analysis with the awareness of this drawback, and thus as a complement of other techniques, such as static analysis [8, 19], code coverage analysis [3, 7], or unit testing [2]. Indeed, some properties, such as the exclusivity property, might be too costly or impossible to analyze statically.

5. Related Work

There is many works on dynamic analysis and program understanding for object-oriented programming languages. We present only the most closely related work and then discuss some other approaches.

Java `PATHFINDER` [4, 29] is a verification and testing environment that integrates model checking, static and runtime program analysis, and testing. It is a powerful environment, which its authors used on several real-life error-critical programs. This environment focuses on the Java programming language in its whole, rather than on a special language. It offers a custom-built JVM, the `JVMJPF`, which executes the program under analysis, and a search component that guides the execution. In `CAFFEINE`, we choose to stay compatible, as much as possible, across platforms and `J2SDK`

versions, thus we use the JPDA, possibly sacrificing performances.

QDBOO [22] allows to specify relationship between instances in C++. Relationships are expressed as universally quantified predicates that can be checked explicitly (with the help of user annotations) or implicitly (as soon as an instance is modified). This approach is more declarative than CAFFEINE (because of the universal quantifier, whereas Prolog only offers existential quantifiers), however predicates deal only with state information (no data or control flow). Moreover, user has no control on the cost of the checks, whereas in CAFFEINE, filters reduce the space and time complexity (in a limited extent).

We get much inspiration from Mireille Ducassé's work on COCA [10] and OPIUM [11]. COCA and OPIUM are programmable debugging systems for C and Prolog, respectively. They are based on a trace model, which is a history of execution events. In OPIUM, the history is considered fully available at any time, although techniques are presented to reduce the required storage space; in COCA, the trace does not require any storage: The analysis is done on the fly. The trace query language for both systems is Prolog, with a set of specialized primitives. In CAFFEINE, we deal with the object model of the Java programming language, which requires a fair amount of work to generate appropriate events.

Our tool comes as a complement of documentation-based search systems, such as I-DOC [21]. I-DOC improves the user-interaction with the documentation and the source code by providing hyper-links and query-based search capabilities. While I-DOC focuses on improving the user's understanding of the static information about the program, CAFFEINE aims at improving the user's understanding of the running program.

CAFFEINE is a programmatic tool, other approaches study visualization techniques and GUIs to help in understanding and analyzing programs, such as ZSTEP 94 [23]. ZSTEP 94 is a debugging environment for Common Lisp. This environment provides the user with a bidirectional *video recorder* interface. Using the video recorder, the user can play forward *and* backward a program, and can access the values of the variables. It also provides mechanisms to play the program backward or forward until some significant change in the graphic output occurs. ZSTEP 94 greatly eases the understanding of the program behavior. However, a maintainer still must verify her conjectures about the program behavior by "hand". Also, queries are not extensible and have a fixed granularity.

CAFFEINE also relate to research on the Java platform debug architecture, such as [9] and [24], on aspect-orientation and the JPDA, such as [26], and on reflective Java extension, such as [30]. While we go further than these approaches with respect to technical issues (for instance, we develop a solution to match the OCL result keyword, as mentioned in [24]), reflective systems are an alternate solution for the implementation of parts of CAFFEINE. However, the ease of implementation of the whole system, its capabilities, and its performances are future research.

6. Future Work

In this paper, we introduce CAFFEINE, a tool for dynamic analysis and understanding of Java programs. We present the trace model and the execution model, corresponding to the Java programming language object-model. We describe the tool architecture, based on the Java platform debug architecture, and implementation issues related to returned values, finalizers, program end, unique identifiers, and our novel solutions.

We also sketch two real-life examples of Java programs analysis of the Singleton design pattern in JHOTDRAW v5.2, and of the composition relationship in the JUNIT v3.7 framework.

The current status of CAFFEINE suggests many extensions. First, we must improve performances, for instance we could replace the generic JDI by a specialized instrumentation of the Java program to generate only interesting events. We also consider using a modified JVM to steady the event generation (*eg.*, finalizers).

Second, we want to provide libraries of Prolog predicates specialized for the analyzes of object-oriented programs. In particular, we are currently developing a language to express interaction diagrams as high-level Prolog predicates. This language shall help maintainers to verify the adequacy between a program behavior and its design, and shall point out the defects in the program behavior *w.r.t.* its documented interaction diagrams. We also plan to introduce universal quantifiers to check dependencies among instances, and contracts on states of instances and method post-/pre-conditions.

Finally, we do not develop our tool in isolation. We currently integrate CAFFEINE with a set of tools to instantiate and to detect design patterns [1]. Indeed, conjectures about the dynamic behavior of a program provide more high-level information, and improve the search for design patterns.

References

- [1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In D. Richardson, M. Feather, and M. Goedicke, editors, *Proceedings of ASE*, pages 166–173. IEEE Computer Society Press, November 2001.
- [2] K. Beck. *Extreme Programming Explained: Embraced Change*. Addison-Wesley, 1999.
- [3] B. Bezier. *Software Testing Techniques*. Van Nostrand Rheinhold Company, New York, 1990.
- [4] G. Brat, K. Havelund, S.-J. Park, and W. Visser. Java PathFinder – A second generation of a Java model checker, July 2000. Workshop on Advances in Verification.
- [5] S. Chiba. Javassist – A reflection-based programming wizard for Java. In J.-C. Fabre and S. Chiba, editors, *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA '98*. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan, October 1998.
- [6] U. Chirico. JIProlog, April 2002. Available at: <http://www.ugosweb.com/jiprolog>.
- [7] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transaction on Software Engineering*, 15(11):1318–1332, November 1989.
- [8] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera specification language. Technical Report KSU CIS Technical Report 2001-04, Kansas State University, 2001. Submitted for journal publication.
- [9] M. Dimitriev. HotSwap technology application for advanced profiling. In *Proceedings of the International Workshop on Unanticipated Software Evolution*, June 2002. Available at: <http://joint.org/use2002/sub/>.
- [10] M. Ducassé. Coca: A debugger for C based on fine grained control flow and data events. In D. Garlan and J. Kramer, editors, *Proceedings of ICSE*, pages 504–513. ACM Press, May 1999.
- [11] M. Ducassé. OPIUM: An extendable trace analyser for Prolog. In A. Bossi and Y. Deville, editors, *The Journal of Logic Programming, Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, volume 41. Elsevier – North Holland, November 1999.
- [12] E. Gamma. JHotDraw, 1998. Available at <http://members.pingnet.ch/gamma/JHD-5.1.zip>.
- [13] E. Gamma and K. Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [15] M. Greenwood. *CFParse Distribution*. IBM Alpha-Works, September 2000.
- [16] Y.-G. Guéhéneuc. Caffeine, May 2002. Available at: <http://www.yann-gael.gueheneuc.net/>.
- [17] Y.-G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe. Bridging the gap between modeling and programming languages. Technical Report 02/09/INFO, École des Mines de Nantes, July 2002.
- [18] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. Technical Report 02/07/INFO, École des Mines de Nantes, May 2002.
- [19] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Proceedings of TACAS*, volume 2280, pages 342–356. Springer-Verlag, April 2002.
- [20] P. V. Hentenryck and M. Dincbas. Forward checking in logic programming. In J.-L. Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference (Volume 1)*, pages 229–256. MIT Press, Cambridge, MA, 1987.
- [21] W. L. Johnson and A. Erdem. Interactive explanation of software systems. In *Proceedings of the Knowledge Based Software Engineering Conference*, pages 155–164. IEEE Computer Society, November 1995.
- [22] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings of ECOOP*, pages 135–160. Springer-Verlag, June 1999.
- [23] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In I. R. Katz, R. L. Mack, L. Marks, M. B. Rosson, and J. Nielsen, editors, *Proceedings of CHI*, pages 480–486. ACM Press, May 1995.
- [24] D. J. Murray and D. E. Parson. Automated debugging in java using OCL and JDI. In M. Ducassé, editor, *Proceedings of the 4th Workshop on Automated Debugging*, August 2000.
- [25] Object Technology International, Inc. Eclipse platform – A universal tool platform, July 2001. Available at: <http://www.eclipse.org/>.
- [26] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. ACM Press, April 2002.
- [27] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communication of the ACM*, 31(11):1259–1267, November 1988.
- [28] Sun Microsystems, Inc. Java platform debug architecture, 2002. Available at: <http://java.sun.com/products/jpda/>.
- [29] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In Y. Ledru, P. Alexander, and P. Flener, editors, *Proceedings of ASE*, pages 3–12. IEEE Computer Society Press, September 2000.
- [30] I. Welch and R. Stroud. Kava – A reflective Java based on bytecode rewriting. In R. Raj and Y.-M. Wang, editors, *Proceedings of USENIX Conference on Object-Oriented Technology*, January 2001.