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
{guehene|douence|jussien}@emn.fr

May 16, 2002

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 only has 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 examples based on the Singleton design pattern and on the composition relationship.

1 Motivation

Soloway *et al.* [21] characterize the software understanding activities as composed of inquiry episodes, during which the 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 [16], 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 the program to understand the program behavior. We develop CAFFEINE [12], a tool that helps a maintainer to check her conjectures about a program, and to understand the correspondence between the static code of the program and its dynamic execution.

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 how a maintainer expresses and verifies her conjecture about a program behavior.

The n-queens problem
Given a $n \times n$-size chessboard and n queens, the algorithm computes the places that the queens must occupy so that they do not attack one another.

The maintainer inherited a Java implementation of the algorithm. After having read the code and understood the overall behavior of the algorithm, the maintainer focuses on the algorithm main loop:

Main loop of the algorithm
The main loop of the algorithm is implemented in the method `placeNextQueen(int column)`, as follows:

```
 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                      // Backtrack.
                        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 16) and it tries again in the same column with the next row (statement `row++`, line 19).

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,
5      query(N1, M).
6  query(N, N).
```

The maintainer instructs CAFFEINE to recursively increment 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 (immediate backtrack) a queen just after having placed her, w.r.t. the total number of backtracks. She turns to CAFFEINE for an answer.

Counting the number of immediate backtracks

The number of immediate backtracks corresponds to the number of time the algorithm control flow matches the pattern:

- Control flow enters in method `setQueen()`.
- Control flow enters in method `placeNextQueen()`.
- Control flow enters in method `removeQueen()`.

The verification of the pattern translates into the query:

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

```
    nextPlaceNextQueen :-
        nextMethodEntryEvent(METHODNAME),
        isPlaceNextQueen(METHODNAME).
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 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 predicate `nextMethodEntryEvent(METHODNAME)`; `METHODNAME` must correspond to method `placeNextQueen()`, line 11: If `METHODNAME` corresponds to method `removeQueen()` or `setQueen()`, lines 12 and 13, the current control flow does not match the expected pattern and the search resumes from the beginning; If `METHODNAME` 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 of the maintainer's question, with the previously presented algorithm is 42: The algorithm removes a queen just after having placed her 42 times. The maintainer could suppress these immediate backtracks by using a look-ahead algorithm [15], with a look-ahead of 1.

We now detail our tool (Section 2): Its trace model and its execution model. Then, we present the implementation of our tool: Its architecture, some interesting technical issues, and performance measurement (Section 3). We also present two examples (Section 4): A conjecture about the binary relationship among classes in a simple constraint solver, and a conjecture about the Singleton design pattern in JHOTDRAW v5.2. 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 describes the content of the trace model, i.e., what are the events generated when analyzing a program.

2.1　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. 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 high-level pattern-matching capabilities and because it is expressive enough, through its backtrack mechanism. Prolog already showed its adequacy to query traces in different works, such as [7, 8].

There are two related predicates to control the program being analyzed:

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

The first predicate, `nextEvent/2`, tells the program to run until the next interesting event and to unify this event with the variable `E`. We specify the interesting events dynamically as we request the next events.

The second predicate, `nextEvent/1`, tells the program to proceed until the next interesting event and to unify this event with the variable `E`. We specify statically the event in which we are interested, when we begin the analysis session. We define `nextEvent/1` using `nextEvent/2`, as: `nextEvent(E) :- nextEvent([], E)`.

2.2   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 predicates, in the form of a functor and a set of parameters. Table 1 presents the list of all the possible events generated by a program execution, and 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 JVM, which results in challenging implementation issues, as we show in this section.

3.1   Architecture

The execution of a program written with the Java programming language takes place in a Java Virtual Machine (JVM), which in turn runs on the operating system of a computer. This layered architecture eases the debugging of Java

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.

programs. Indeed, the JVM provides a standard interface to debug the program that it executes, the Java Platform Debug Architecture (JPDA) [22].
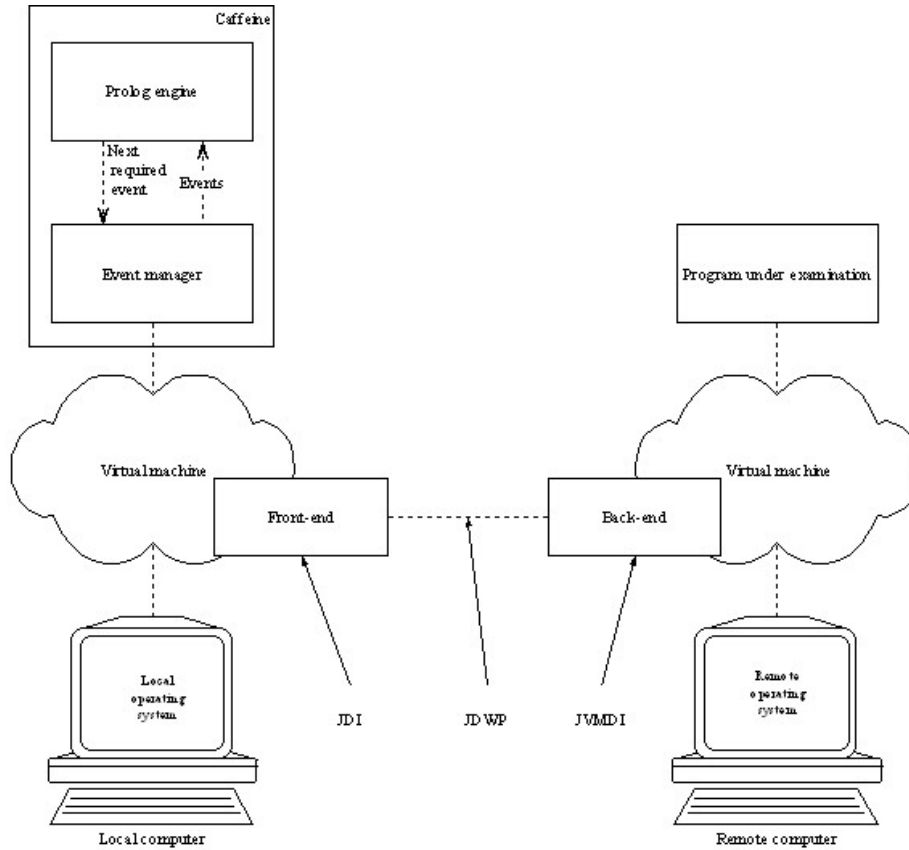

Figure 1: The Java Platform Debug Architecture

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 enables the connection with an already existing remote JVM or the creation of a remote JVM, and the interaction with it. The JDI dialogs with the remote JVM running on the remote computer through the specialized Java native interface for third-party debugging tools, the Java Virtual Machine 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 `Virtual-Machine` interface, the local program receives events and has access to instances,

7

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 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 [4]. JIProlog is a Prolog engine implemented in Java, offering 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/2` predicate as a custom predicate. This custom predicate resumes the execution of the remote program, and obtains the next desired events generated by the remote JVM, as specified by the first argument of the predicate, through the `EventManager` class.

Finally, a `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 proper query and starts the resolution of the query.

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

```
 1  package caffeine.example.queens;
    import caffeine.monitor.prolog.Caffeine;

    public final class NQueensCaffeineLauncher {
 5      public static void main(final String[] args) {
            Caffeine.run(
                "Caffeine/Example/Queens/Rules.pl",
                <Classpath omitted here>,
                "caffeine.example.queens.NQueens",
10              new String[] { "caffeine.example.queens.*" },
                Caffeine.SystemClasses,
                Caffeine.GenerateMethodEntryEvent,
                null);
        }
15  }
```

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

- Line 7: The name of the file, which contains the Prolog query to verify.

- Line 8: The program execution *classpath*.

- Line 9: The main class of the program to analyze.

- Line 10: The event positive filters.

- Line 11: The event negative filters (only used if no positive filter is provided).

- Line 12: The event required for the analysis. In our example, the analysis only requires the program to generate events for method entries, to answer the maintainer's conjecture about the n-queens algorithm.

- Line 13: A list of fields to monitor, the value `null` indicates CAFFEINE that no field needs monitoring.

The following is the partial output generated when running the `NQueens-CaffeineLauncher` 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. The solution appears in the Prolog query last output, `I removed a queen after having placed her for the 42 time(s)`, and in the result of the query, `Solution = main(0, 42)`:

```
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 1 time(s)
<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
(Remote JVM) (With 105 backtracks.)
Solution = main(0, 42)
JVM running time: 138890 ms.
(37 ms. per step for the 3662 steps.)
```

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

3.2 Returned value

By definition, the method exit event generated by the remote JVM does not provide the value being returned by non-`void` methods. According to SUN MICROSYSTEMS, INC. [14], a future release of the J2SDK will fix this limitation in the API.

We overcome this limitation by wrapping any returned value in a special method, at load-time and at the byte-codes level[1], using CFPARSE [11]. 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 following `becomeOlder()` method, which returns an integer:

---

[1] For the sake of clarity, in this section, we present the transformations at the source-code level.

```
1  int becomeOlder() {
       this.age = Util.getCurrentDate() - this.birthDate;
       return this.age++;
4  }
```

We wrap the value returned by the method before returning it. The following code pictures the new method implementation:

```
1  int becomeOlder() {
       this.age = Util.getCurrentDate() - this.birthDate;
       return
           CaffeineMethodReturnedValueWrapper.
5              methodReturnedValueWrapper(this.age++);
6  }
```

The wrapper method directly forwards its parameter as returned value:

```
1  public abstract class CaffeineMethodReturnedValueWrapper {
       public static final int methodReturnedValueWrapper(
           final int value) {
           return value;
       }
5      <Other methods omitted.>
6  }
```

We introduce a method call for each return. This technique based on a wrapper is a solution to obtain the returned value.

3.3   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 value)` method implemented by the `System` class, although this method is deprecated [23]. 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 finalizer()` 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 required, a finalizer to loaded classes, using JAVASSIST [3]. The added finalizer delegates its operation to its `super` finalizer.

However, these strategies does 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.

- Ensures a partial order among the calls to finalizers when the JVM exits.

Reference counting is a candidate technique for such a garbage-collector, the Java Native Interface is a candidate technique for the implementation.

3.4   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 can use neither of these strategies within CAFFEINE.

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 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 throws a `NullPointerException`.

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 limitation[2] prevents the garbage-collector to collect the instance after a remote method invocation, and thus prevents the instance finalizer to execute and the corresponding event to happen.

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

3.5   Performances

Table 2 shows some performance results for the n-queens implementation in Java. We measure those results on a Pentium-III processor, with 256 Mb of RAM, Windows 2000, running CAFFEINE and the example with the J2SDK v1.4.0 in the ECLIPSE [20] development environment.

These measures show that the JVM event generation mechanisms dramatically slow down the program execution, even using the filter-capabilities provided by the JDI. The execution time increases by a factor of about 300, between a normal execution and an execution with the JVM generating events[3]. In comparison, the analysis does not significantly slow down the overall execution.

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

---

[2]The authors notice this limitation for the JPDA implementation of the J2SDK v1.3.1 and J2SDK v1.4.0.

[3]The interested reader may find more measures on the performance of the JPDA in [2].

Program	Time (in seconds)	Increase Factor
Stand-alone	0.400	
Debug mode		
Without events	0.400	1.000
With events	120.134	300.335
Analysis		
Only required events	125.645	1.046

Table 2: Performances and relative time increase factor for the Java n-queens algorithm

4 Examples of Dynamic Analyses with Caffeine

We now sketch two real examples of conjecture about two different Java programs and the solutions CAFFEINE offers to verify them. For lack of space, we only present the problems and briefly describe the solutions using CAFFEINE. The first conjecture concerns the implementation of the Singleton design pattern in JHOTDRAW v5.2. The second conjecture concerns the relationship between two classes in the CACAO constraint solver.

4.1 Singleton design pattern

The JHOTDRAW v5.2 [9] 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 [10]. In particular, the designers use the Singleton design pattern in several places.

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. 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 time the `IconKit` class is instantiated.

After running a typical session with JHOTDRAW 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.

4.2 Binary Class Relationships

We want to represent the CACAO constraint solver [5, 6], using the UML visual modeling language. In the UML, there exist three different binary class relationships:

- Association: A link between two classes.

- Aggregation: A link between two classes, respectively the *whole* and the *part*. It is an anti-symmetric association relationship. The part cannot have an aggregation relationship with the whole, but may have an association link. Conceptually, a part has no sense without a whole.

- Composition: An aggregation relationship where the parts held by the whole are destroyed when the whole is destroyed. These parts can be exchanged during the life-cycle of the whole, but all the parts owned by the whole at the moment of its deletion are destroyed in cascade.

We conjecture that the `fr.emn.info.cacao.Relation` class is in composition relationship with the `fr.emn.info.cacao.Variable` class. It is difficult to verify statically this relationship because it involves the lifetimes of the instances of the two classes.

We can express our conjecture in term of field modifications and finalizers to check the lifetimes and the instance-sharing property [13] of the instances of class `Variable` w.r.t. instances of class `Relation`. We run CACAO with CAFFEINE to solve the n-queens problem using constraints and, indeed, we find that the two classes are in composition relationship.

5   Related Work

There is few work on dynamic analysis and program understanding for object-oriented programming languages. We first present closely related work and then discuss some other approaches.

QDBOO [17] allows to specify relationship between instances in C++. Relationships are expressed as universally quantified predicates which 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), however predicates deal only with state information (no data or control flow). Moreover, user has no control on the cost of the checks.

We get much inspiration from Mireille Ducassé's work on COCA [8] and OPIUM [7]. 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 technics 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 [16]. 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 [18]. 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 [19], on aspect-orientation and the JPDA, such as [2], and on reflective Java extension, such as [24]. 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 [19]), 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 propose CAFFEINE, a tool for dynamic analysis and understanding of Java programs. We introduce the trace model and the execution model, based on the Java programming language object model. Then, we present the tool architecture based on the Java platform debug architecture, and implementation issues related to returned values, finalizers, unique identifiers, and our novel solutions.

We also sketch two real examples of Java programs analysis based on the Singleton design pattern in JHOTDRAW v5.2, and the binary class composition relationship in the CACAO constraint solver.

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.

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.

Third, we also consider using a modified JVM to steady the event generation (eg., finalizers) and to improve performances.

Fourth, we must study the downside of relying on dynamic analysis and study code coverage problems and possible ways to improve the reliability of our analyzes.

Finally, we do not develop our tool in isolation. We currently integrate CAFFEINE with the set of tools we presented last year to instantiate and to detect design patterns [1]. Indeed, conjectures about the dynamic behavior of a program provide more high-level pieces of information, thus improving 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] A.Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, April 2002.

[3] S. Chiba. Javassist a reflection-based programming wizard for Java. *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98*, 1998.

[4] U. Chirico. JIProlog, April 2002. Available at: `http:// www.ugosweb.com/ jiprolog`.

[5] R. Douence and N. Jussien. Non-intrusive constraint solver enhancements. Technical Report 02-2-INFO, École des Mines de Nantes, 2002.

[6] R. Douence and N. Jussien. Non-intrusive constraint solver enhancements. In *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Enschede, The Netherlands, April 2002. University of Twente.

[7] M. Ducassé. OPIUM: An extendable trace analyser for prolog. Technical Report 3257, INRIA, September 1997.

[8] M. Ducassé. Coca: A debugger for c based on fine grained control flow and data events. Technical Report 3489, INRIA, September 1998.

[9] E. Gamma. JHotDraw, 1998. Available at `http:// members.pingnet.ch/ gamma/ JHD-5.1.zip`.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] M. Greenwood. *CFParse Distribution*. IBM AlphaWorks, September 2000.

[12] Y.-G. Guéhéneuc. Caffeine, May 2002. Available at: `http://www.yann-gael.gueheneuc.net/ Work/ Research/ Caffeine/ Download/`.

[13] Y.-G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe. Bridging the gap between modeling and programming languages. Technical report, École des Mines de Nantes, May 2002. To appear.

[14] Y.-G. Guéhéneuc and T. Bell. Question regarding the access to the StackFrame, October 2001. Personal e-mail available upon request at `guehene@emn.fr`.

[15] 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.

[16] 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.

[17] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In *Proceedings of ECOOP*, 1999.

[18] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of CHI*, pages 480–486. ACM Press, 1995.

[19] 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.

[20] I. Object Technology Internationl. Eclipse platform – a universal tool platform, July 2001. Available at: `http:// www.eclipse.org/`.

[21] 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.

[22] I. Sun Microsystems. Java platform debug architecture, 2002. Available at: `http:// java.sun.com/ products/ jpda/`.

[23] I. Sun Microsystems. System.runFinalizersOnExit(boolean), May 2002. Available at: `http:// java.sun.com/ j2se/ 1.4/ docs/ api/ java/ lang/ System.html# runFinalizersOnExit(boolean)`.

[24] I. Welch and R. Stroud. Kava - a reflective java based on bytecode rewriting. In *Proceedings of USENIX Conference on Object-Oriented Technology*, 2001.