Science of Computer Programming 94 (2014) 362–378

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Monitoring method call sequences using annotations ☆

B. Nobakht [a,b,*], F.S. de Boer [a,c], M.M. Bonsangue [a,c], S. de Gouw [a,c], M.M. Jaghoori [c]

[a] Leiden Institute of Advanced Computer Science, Universiteit Leiden, The Netherlands
[b] SDL Fredhopper, Amsterdam, The Netherlands
[c] Centrum Wiskunde and Informatica, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 30 October 2011
Received in revised form 12 September 2013
Accepted 11 November 2013
Available online 4 December 2013

Keywords:
Object monitoring
Runtime verification
Method call sequence specification
Code annotation
Component-based testing

ABSTRACT

In this paper we introduce JMSeq, a Java-based tool for monitoring sequences of method calls. JMSeq provides a simple but expressive language to specify the observables of a Java program in terms of sequences of possibly nested method calls. Similar to many monitoring-oriented environments, verification in JMSeq is done at runtime; unlike all other approaches based on aspect-oriented programming, JMSeq uses code annotation rather than instrumentation, and therefore is suitable for component-based software verification.

1. Introduction

Monitoring refers to the activity of tracking observable information from an execution of a system with the goal of checking it against the constraints imposed by the system specification. The observable information of the monitored system typically includes behavior, input and output, but may also contain quantitative information. A monitoring framework consists of a monitor that extracts the observable behavior from a program execution, and a controller that checks the behavior against a set of specifications. Should an execution violate the constraints imposed by the specification, corrective actions can be taken.

What can be monitored depends, of course, on what can be observed in a system. When the application source code is available, its code can be instrumented to produce informative messages about the execution of an application at run time. New code is inserted for the management of monitoring and verification mechanisms, while preserving the original logic of the application. Bytecode instrumentation is another technique widely used for monitoring and tracing execution of Java programs. Bytecode instrumentation consists of inserting bytecode into classes either at compile-time or at runtime. It is typically preferred to source code instrumentation because it can be used to enhance a program temporarily for debugging or analysis purposes, and no source code is required. Main disadvantages however of bytecode instrumentation are the following:

☆ This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu/).

* Corresponding author.
E-mail addresses: bnobakht@liacs.nl (B. Nobakht), frb@cwi.nl (F.S. de Boer), marcello@liacs.nl (M.M. Bonsangue), cdegouw@cwi.nl (S. de Gouw), jaghoori@cwi.nl (M.M. Jaghoori).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.030

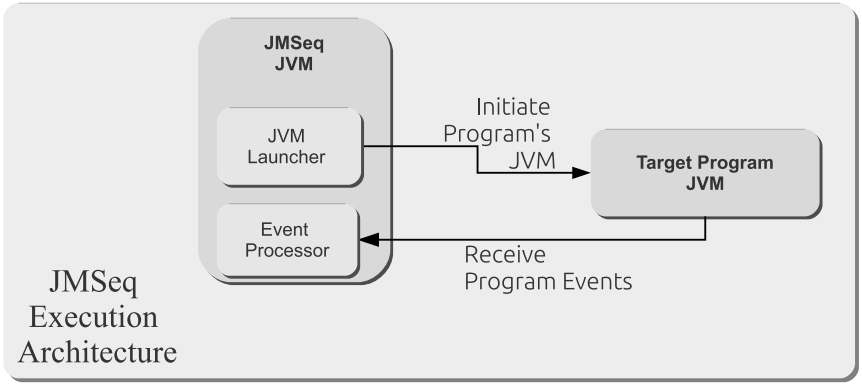

Fig. 1. JMSeq execution architecture.

- Although there exist tools for facilitating bytecode instrumentation, building instrumentation in bytecode requires an in-depth knowledge of bytecode; thus, it is *error-prone*.
- Instrumentation by its very nature is an intrusive approach; i.e., the target program is modified either at source level or bytecode level. This makes it unsuitable, for example, for monitoring classes in the Java standard library, which are supposed to remain unchanged. Moreover, if two different bytecode agents simultaneously instrument the same program, they must be compatible; otherwise, the resulting bytecode may contain *conflicts*.
- From the perspective of business security, a verification approach that modifies a program potentially increases its *vulnerability* and as such is of restricted applicability.

In this paper, we introduce JMSeq, a tool for monitoring sequences of method calls in Java programs. JMSeq is based on Java annotations rather than (source code or bytecode) instrumentation. Therefore, it does *not* require access to the source code of the target application. JMSeq detects the annotated methods from the target application at runtime. Furthermore, Java annotations themselves have no direct effect on program execution or logic of the application; they carry metadata that can optionally be used at runtime to perform operations orthogonal to the execution of the main application. The main distinguishing overall characteristic of JMSeq is the integration of the following main features.

Java annotations. JMSeq annotations specify method call sequences in terms of regular expressions of method call signatures. This pure declarative language is simple and intuitive in specification of different types of sequences of method calls.

JVM decoupling. To ensure that the target program is in *no* way affected or changed during monitoring, JMSeq runs in a separate JVM from the target application's JVM. JMSeq uses Java Platform Debugger API (JPDA) [3] to initiate a separate JVM and verify the target application using events from the target JVM. Fig. 1 displays the high-level execution architecture of JMSeq.

Fail-fast approach. JMSeq terminates execution as soon as a violation of a specification occurs. This approach prevents unsafe behavior, enables the localization of an error, and facilitates a reaction to a failure immediately, thus making the software more robust [28].

An important consequence of these features is that neither the source code (even if available) nor bytecode is modified by JMSeq. Furthermore, JMSeq does not require source code to be present. As such, JMSeq is suited for the specification and verification of both newly built software and legacy systems, including third-party libraries (and even the Java standard library). For example, given the API of a legacy system, its expected behavior can be specified by annotating the methods of a "driver" program which triggers the legacy code. JMSeq then verifies the triggered behavior of the legacy code using these annotations. In the context of an ongoing software development process, JMSeq can also be used as mentioned above by means of a "driver" program but also by directly annotating the "interfaces" of the system API. By annotating the interfaces of a system any service object that implements the interface behavior inherits the annotations and can be verified by JMSeq. In this respect, JMSeq also introduces integration with frameworks such as JUnit (cf. Section 5.2).

The rest of the paper is organized as follows. The first three sections describe the usage of JMSeq. Section 2 presents the language for specifying sequences of method calls, and Section 3 discusses the types of annotations available in JMSeq. These two sections are connected via examples in Section 4. Afterwards, implementation of the JMSeq framework is explained in detail in Section 5. The practical applicability of JMSeq is demonstrated by means of a case study from industry in Section 6 and performance evaluation in Section 7. Finally, related work is discussed in Section 8. Section 9 concludes the paper.

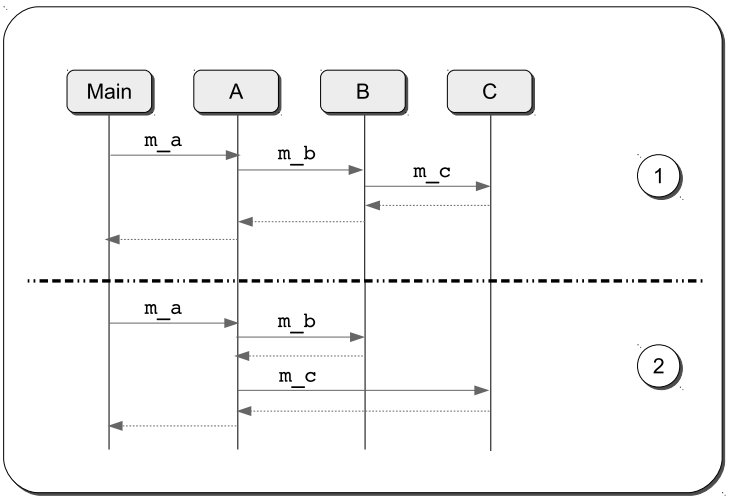

Fig. 2. Examples of method call sequence specification.

$$
\begin{aligned}
\langle Specification\rangle ::= {} & \langle Call\rangle \mid \langle Call\rangle\ \langle Specification\rangle \mid \\
& \langle Specification\rangle^{\mathrm{m}} \mid \langle Specification\rangle\$ \mid \langle Specification\rangle\# \mid \\
& (\langle Specification\rangle) \\
\langle Call\rangle ::= {} & \{\texttt{call}(\langle Signature\rangle)\langle InnerCall\rangle\}\$ \mid \\
& \{\texttt{call}(\langle Signature\rangle)\langle InnerCall\rangle\}\# \mid \\
& \{\texttt{call}(\langle Signature\rangle)\langle InnerCall\rangle\}^{\mathrm{m}} \mid \\
& \{\texttt{call}(\langle Signature\rangle)\langle InnerCall\rangle\} \mid \\
& \{\texttt{call}(*)\} \mid \\
& < \langle Call\rangle\ ?\ \langle Call\rangle > \\
\langle InnerCall\rangle ::= {} & [\langle Call\rangle]\langle InnerCall\rangle \mid \epsilon \\
\langle Signature\rangle ::= {} & \langle \textit{AspectJ Call Expression Signature}\rangle
\end{aligned}
$$

Fig. 3. Method sequence specification grammar.

2. Method call sequence specification

We consider a component to be a collection of compiled Java classes. The relevant dynamic behavior of a component can be expressed in terms of specific sequences of messages among a small, fixed number of objects. In Fig. 2, we see two UML message sequence diagrams each describing how, and in which order, the four objects interact with each other. In this section, we develop a specification language for describing these kinds of interactions.

In addition to specifying the order, a specification language for sequences of method calls needs to distinguish between the following two cases (depicted in Fig. 2).

Case 1 shows a scenario in which (the call to) `m_c` is nested in `m_b` since `m_c` is called during the activation of `m_b` (i.e., after `m_b` is called and before it returns). Similarly, both `m_b` and `m_c` are nested in `m_a`.

Case 2 represents a method call in which methods from different/same objects are called in a sequential rather than nested manner. In this example, both `m_b` and `m_c` are called by `m_a`.

Typically, a program will need a combination of both cases to specify its dynamic behavior. Specifying only the order of the method calls is not enough, as both cases above have the same order of method calls. It is thus required to have a specification technique that distinguishes between the *method calls* and *method returns*.

In JMSeq, a specification is written as a post-condition associated with a method. It specifies the set of possible sequences of method calls, or protocol, of an object in the context of the relevant part of its environment. Formally, sequences of method calls can be specified by means of the grammar in Fig. 3.

A *specification* consists of a sequence of calls that can be repeated an a priori fixed number of times ("$\langle$*Specification*$\rangle^{m}$"), with $m \geqslant 0$, one or more times ("$\langle$*Specification*$\rangle$\$"), or zero or more times ("$\langle$*Specification*$\rangle$#"). Although JMSeq does not use aspect-oriented programming in its implementation, we have used the generic method call join points syntax of AspectJ [20], using for example # to denote the more standard Kleene star operation. Additionally, ($\langle$*Specification*$\rangle$) is a way to group specifications to avoid ambiguity. To improve readability of the specifications, grouping is only used when necessary.

A *call* is a call signature followed by a (possibly empty) sequence of inner calls. Each call can be repeated either one or more times, zero or more times, or exactly m-times, for some $m \in \{1, 2, 3, \ldots\}$; i.e., m is a constant. Having the repetition by m does not make the language go beyond the expressiveness of regular expressions. Additionally, the wild card {`call(*)`} denotes a call to an arbitrary method. To support branching, JMSeq also provides $< \langle Call \rangle ? \langle Call \rangle >$ to allow the specification of a choice in a sequence of method executions.

Inner calls are calls that are executed before the outer call returns, i.e., they are nested. We do not have explicit return messages, but rather we specify the scope of a call using brackets. Information about the message call, like the type of the caller and the callee, method name, and the types of the parameters are expressed using a syntax similar to that of AspectJ. Simply put, a call signature is of the form

```
call([ModifiersPattern] TypePattern
        [TypePattern . ] IdPattern (TypePattern | ".." , ... )
        [ throws ThrowsPattern ]
)
```

reflecting the method declarations in Java. The two leftmost patterns specify the method modifiers (like “static” or “public”) and return types. Here `IdPattern` is a pattern specifying the method name. It can possibly be prefixed at the left by a specification of the type of the object called. The method name is followed by a specification of the types of the parameters passed during the call, and possibly by a specification of the exceptions that may be thrown. It implies that JMSeq specification grammar can distinguish the overloaded methods in a class. Patterns may contain wild card symbols “`*`” and “`..`”, used for accepting any value and any number of values, respectively. For example, the call

```
call(* *.A.m_a(..))
```

is denoting a call to method `m_a` of any object of type `A` that is placed in any package, and returning a value of any type. If there is a need to be more specific, a possible restatement of the same specification could be:

```
call(int nl.liacs.jmseq.*.A.m_a(Object, double))
```

Next we give a few examples of correct method sequence specifications. For instance, the specification of the sequence in case 1 of Fig. 2 is given by:

```
{call(* *.A.m_a(..))[{call(* *.B.m_b(..))[{call(* *.C.m_c(..))}]}]}
```

Here it is important to notice that the call to method `m_b` is internal to `m_a`, and the call to `m_c` is internal to `m_b`. The sequence in case 2 of the same figure would be:

```
{call(* *.A.m_a(..))[{call(* *.B.m_b(..))}][{call(* *.C.m_c(..))}]}
```

where both calls to methods `m_b` and `m_c` are internal to `m_a`.

These two cases depict a fixed sequence of method calls. More interesting are the cases when, for instance, the method `m_a` should be called at least once before any possible method call to `m_b` or `m_c`:

```
{call(* *.A.m_a(..))}$<{call(**.B.m_b(..))}#?{call(* *.C.m_c(..))}#>
```

Such a specification is used in circumstances where the execution of `m_a` is a prerequisite to the execution of either of `m_b` or `m_c`.

It is notable that the actual specifications are regular expressions; for example, they may contain unbounded repetitions and a choice of calls.

3. Annotations for method call sequences

Java enables developers to attach extra metadata to their programs using annotations. These metadata do not directly affect the program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the behavior of the running program [2]. Annotations can be read from source files, class files, or reflectively at runtime. One can define custom annotation types and use them to provide the desired types of metadata for the declarations in his program. In Java, an annotation is a special kind of modifier, and can be used anywhere other modifiers (such as `public`, `static`, or `final`) can be used.

We use annotations as an enabling feature to test components without the need to have their source code. We only use the metadata loaded from the annotations during runtime. These metadata include the specification of the sequences of method calls based on the grammar discussed in Section 2. The metadata is placed on classes or methods in a Java program. Then, they are loaded into JMSeq when it monitors the actual observable behavior of the program at runtime.

JMSeq defines two type of annotations: sequenced object annotations and sequenced method annotations.

Listing 1 SequencedObject annotation declaration.

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.TYPE)
3 public @interface SequencedObject {
4     // we need no properties for this annotation as this is only a marker.
5 }
```

Listing 2 SequencedMethod annotation declaration.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  public @interface SequencedMethod {
4
5      String value();
6
7      boolean allowExceptions() default true;
8
9      Class<?>[] expect() default {};
10
11     Class<VerificationFailureHandler> verificationFailureHandler();
12 }
```

3.1. Sequenced object annotations

To verify the sequence of executions, methods in classes need to be annotated. Thus, JMSeq needs to detect all classes that contain some method carrying an annotation for verification. To improve readability and performance, JMSeq introduces `SequencedObject` annotation. This annotation is a marker for such classes and notifies the annotation metadata loader (cf. Section 5.1.2) that the objects instantiated from the annotated class contain methods that specify a sequence of method executions. The code is given in Listing 1.

`@Retention(RetentionPolicy.RUNTIME)` declares that this annotation is only applicable during runtime and may not be used in other scenarios. Moreover, `@Target(ElementType.TYPE)` declares that this annotation can only be used on types including classes, interfaces and enumerated types; in other words, this annotation may not be used to mark a method in a Java class.

3.2. Sequenced method annotations

A sequenced method annotation is used to specify the sequences of method calls beginning from a given method. The annotation requires a string property declaring the sequence specification discussed in Section 2. Listing 2 presents the declaration of this annotation type.

`@Target(ElementType.METHOD)` declares that this annotation is only applicable to *methods*. The string value from `value()` holds the sequence specification (cf. Section 2). Additionally, a sequenced method annotation allows verification of exceptions during runtime. The tester can configure whether exceptions are allowed or not using the combination of `allowExceptions()` and `expect()`:

`allowExceptions() = true` implies that exceptions are allowed during execution; this is the default case. If there is no exception configured using `expect()`, *any* exception is acceptable; otherwise, an exception is acceptable only if it is specified in the array of classes in `expect()` or if it is a subclass of one of them. If an exception occurs and it is "expected", JMSeq allows the program to continue execution; the program should nevertheless be able to properly deal with the exception.

`allowExceptions() = false` means that no exceptions are expected during the execution. In this case, the value specified by `expect()` is ignored.

JMSeq additionally provides a general mechanism to specify how to handle situations in which the sequenced execution fails to complete. This can happen because of a method call which was not included in the specification, or an unexpected exception. To this end, JMSeq introduces `VerificationFailureHandler` interface as an extension point where the tester or developer can provide custom behavior to gracefully deal with the verification failure. This interface introduces a single method: `handleFailedExecution(Execution e)`. This method receives information about the execution through a parameter of type `Execution` (cf. Section 5). `Execution` is the central data model that carries information about the method execution sequences; the trail of the executions may be useful to the developer to examine the failure causes. If no such handler is provided, verification failure results in termination of program execution with a failure status.

Listing 3 Sample annotated specification.

```
1
2  // Case 1
3  @SequencedObject
4  public class Main {
5
6     @SequencedMethod(
7       "{call(* *.A.m_a(..))[{call(* *.B.m_b(..))" +
8       "[{call(* *.C.m_c(..))}]}]}")
9     public void main() {
10       // ...
11    }
12
13    public void init() {
14       // ...
15    }
16 }
17
18 // Case 2
19 @SequencedObject
20 public class Main {
21
22    @SequencedMethod("{call(* *.A.m_a(..))" +
23      "[{call(* *.B.m_b(..))}][{call(* *.C.m_c(..))}]}")
24    public void main() {
25       // ...
26    }
27
28    public void init() {
29       // ...
30    }
31 }
```

Section 4 provides more discussion along with different examples on using JMSeq.

4. JMSeq by example

In this section, we go through different usage scenarios of JMSeq by means of illustrative examples.

4.1. Sequenced execution specification

In Listing 3, we again go through the example introduced in Section 2. In this example, we annotate the method `main()` with the sequences of method calls describing the two scenarios given in Fig. 2. The JMSeq specifications used here are discussed in Section 2. The listing shows how to use them in a Java program.

4.2. Exception verification

We discuss how JMSeq can be used to verify occurrence of exceptions during execution of an application using the example given in Fig. 2 Case 1.

Let's assume that `m_c` may encounter `NullPointerException` and `m_b` may face `IOException`. But, we do not hold any assumption on the location in the code where the exceptions may be handled or even totally ignored. We use the following annotation in Listing 4 to express this situation.

During the execution of the application, if any exception occurs, JMSeq receives the event of the exception occurrence. Since exceptions are allowed, there will be no verification failure by JMSeq as long as the exceptions received and verified are either instances of the specified exception classes or any of their subclasses. However, if the underlying application does not handle the exceptions (i.e., proper use of `try/catch`), they are propagated up in the call stack and this may result in JVM crash.

For instance, if an instance of `FileNotFoundException` occurs in `m_c`, JMSeq accepts the exception (because it is a subclass of `IOException`) and allows the application to continue. If the application at some point deals with the exception, the execution continues; otherwise JVM stops, complaining that the exception is not handled. On the other hand, if an `IllegalStateException` happens during the execution of the program, JMSeq does not allow the application to

Listing 4 Exception specification for Case 1.

```
1  // Case 1
2  @SequencedObject
3  public class Main {
4
5     @SequencedMethod("{call(* *.A.m_a(..))" +
6       "[{call(* *.B.m_b(..))[{call(* *.C.m_c(..))}]}]}",
7       allowExceptions = true,
8       expect = {NullPointerException.class,
9                 IOException.class})
10    public void main() {
11       // ...
12    }
13
14 }
```

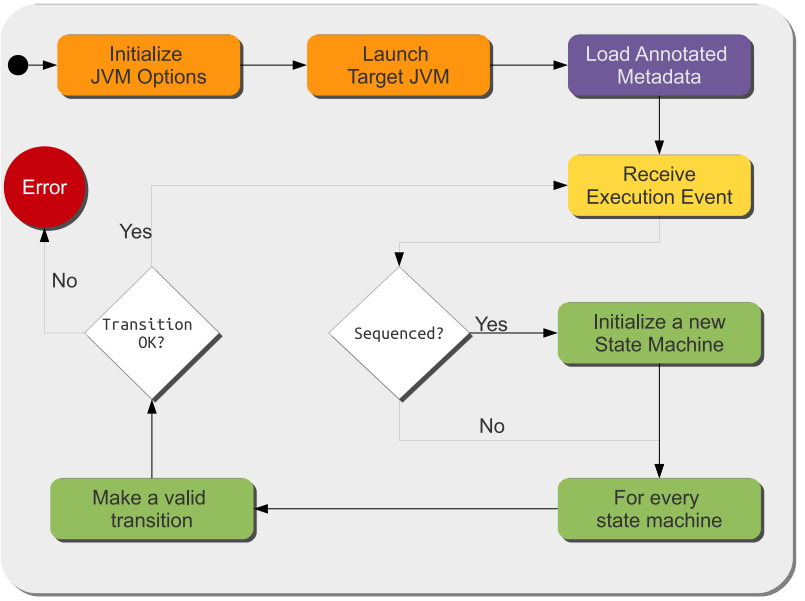

Fig. 4. Overview of JMSeq's process to verify a program.

receive the exception since it is not one of the expected exceptions in the specification. Instead, it runs the failure code provided by the tester (if any).

In another situation in Case 1, assume that *no* exceptions are acceptable during the execution. For this to happen, we simply use `allowExceptions = false` in the specification annotation. In this case, no exception should occur during the execution of the application; otherwise JMSeq issues a verification failure. Besides, when no exceptions are allowed, using `expect()` has no effect on JMSeq verification behavior.

JMSeq additionally supports integration with testing frameworks such as JUnit. An example of such integration is discussed in Section 5.2.

5. The JMSeq framework

In this section, we present the JMSeq monitoring and verification framework and discuss its implementation, which uses Java 5 Annotations [2] and Java Platform Debugger API [3]. More implementation details, including examples and documentation, can be found at http://github.com/nobeh/jmseq.

As discussed above, to use JMSeq, it is assumed that Java components are supplied with sequenced object and sequenced method annotations. Multiple annotations are possible within the same class, but not for the same method. This implies that a component specification is in fact scattered across its constituent classes, each of which with a local and partial view of the component behavior. Only the annotated methods will be monitored by JMSeq. The overall verification process in JMSeq is depicted in Fig. 4.

As a Java program, JMSeq executes naturally inside a Java virtual machine (JVM); JMSeq additionally initiates another JVM (called the *Target JVM*) to be able to control the sequenced execution of the target program. Through initial parameters, the target JVM is configured to report back to JMSeq some events such as method entry and method exit. Furthermore, JMSeq does not verify events from all objects but only those specified in these parameters. The target JVM takes advantage of the Java Platform Debugger Architecture (JPDA) to access the needed details on the execution of the application while it is running.

Before verification starts, JMSeq inspects the classes in the execution class path searching for methods that are annotated for sequenced verification; during this process, it collects all of the available method call sequence metadata. This step (denoted as Load Annotated Metadata in Fig. 4) does not need the source code, but rather uses the annotations metadata available in the compiled code.

The target program subsequently starts its execution and JMSeq reacts to the events it receives. JMSeq uses state machines to verify the method calls occurring in an application during runtime. The state machines are indexed by the pairs of method signature and the callee object; thus JMSeq can uniquely identify the associated state machine. When JMSeq receives a method execution event, it checks if there is a need to initialize a new state machine for that method execution. A state machine is initialized if there is no state machine for the current method execution and the method is annotated `@SequencedMethod`. Then, JMSeq tries to verify the current method execution in the context of all active state machines for other methods. For every state machine, JMSeq tries to make a valid transition in the state machine. Verification will continue if such a transition can be made for all state machines; if not, JMSeq stops the execution reporting a violation of the specification.

The process of matching method execution events with state machine transitions is done in JMSeq by following these steps:

1. First, every event received from the target JVM, together with the information associated to the event, is transformed to an instance of `Execution` to capture various details about the execution of a method. Then, at verification time, the instance of `Execution` is transformed to an instance of `CallExpression` to hold method signature details used by the state machine to make a transition. (Section 5.1.1)
2. Using the metadata available from the annotations of the methods, then, the next "possible call expressions" of the current state, corresponding to the possible next transitions of the state machine, are built. (Sections 5.1.2, 5.1.3)
3. A match making is done between the possible call expressions (state machine transitions) and the current call expression as the candidate. If a match is found, the transition is made and the method is accepted; otherwise, it fails. When a failure occurs, JMSeq will execute the custom verification failure handler that is implemented either as part of the component code by the programmer, or externally by the system tester. (Section 5.1.3)

JMSeq computes the possible call expressions on-the-fly instead of constructing the full state machine in the beginning of the verification process. This way it avoids the construction of states in the state machine that are never used in the execution. To avoid repeated computations of the same states, JMSeq utilizes dynamic programming techniques such as in-memory object caches.

5.1. JMSeq architecture

The overall architecture of JMSeq is given in Fig. 5. It basically consists of three main modules: one for handling the events raised by the JVM executing the target program, another module for storing the annotation information, and a third one to perform the runtime verification.

5.1.1. Event handling module

Java Debugger Interface (JDI) is the interface of Java Platform Debugger Architecture (JPDA) that gives access to different details during the execution of a program. Following the JDI event model, JMSeq takes control over some of the execution events that JVM publishes during a program execution. Therefore, a component was designed to model and hold the execution trace events required for event handling and execution verification.

1. `Execution` is the central data structure that holds all the information about execution that is made available through the JDI event mechanisms. Relevant events include "method entry", "method exit", and "exception" occurrences. It provides access to information such as:
   - The object that is currently executing (*the callee object*) and its unique identifier in JVM.
   - Event details through subclasses such as method return values or *the caller object* reference in case of a method exit event.
   - *Parent execution*: every execution can hold a reference to its parent execution object forming a directed tree of executions. This helps traversing the executions at validation time.
2. `EventHandler` is the event handling interface that is injected into JVM with access to JDI information. The event handler receives the events it is subscribed to and possibly takes an associated action. In particular it creates an instance of `Execution` for each sequence annotation and it stores it in the `ExecutionTraceOracle` registry. In general, all

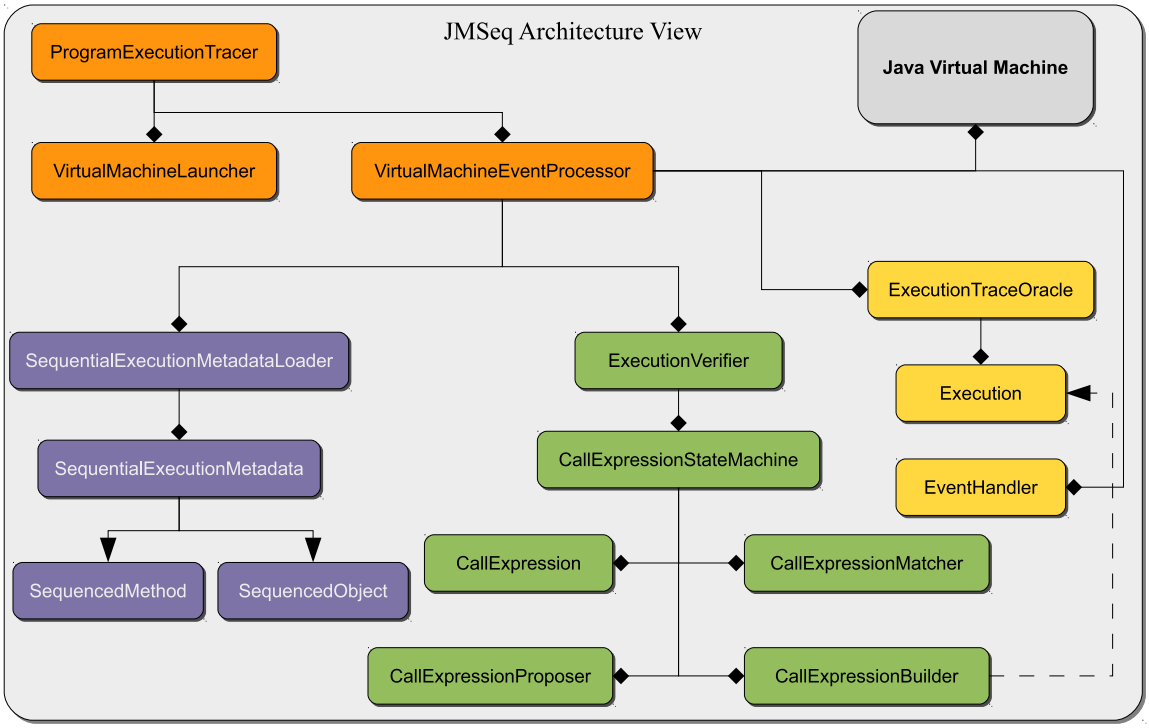

Fig. 5. Software architecture for method sequence specification.

events received and processed by `EventHandler` are stored in this registry for further use by other components, for example, by the verification module.

5.1.2. Annotation metadata module

As the execution traces are stored in a repository, they are supposed to be checked and verified against the formal specifications as described on the `@SequencedObject` and `@SequencedMethod` annotations of the compiled classes in the program. It is the task of the annotation metadata module to store this information. A small utility component in this module is responsible to read and load the metadata of all classes that are annotated. This information is used when there is a need to verify the conformance of an execution event.

5.1.3. Execution verification module

During execution of a program, those events that need to be monitored are received and verified against a message sequence specification. For every specification in the metadata repository, a state machine is created on the fly. At each state, JMSeq computes the Brzozowski derivative of the regular expression associated with the current state and memorizes this computation for further use. This avoids the full construction of the state machine at each step of verification.

When a method execution event is received, either the corresponding state machine exists or it should be created. A state machine is initialized if there is no state machine for the current method execution and the method is annotated `@SequencedMethod`. JMSeq then tries to verify the event against all active state machines. For every state machine, an attempt is made to make a valid transition. If the transition is successfully made, JMSeq continues. Otherwise, an "invalid" transition is detected; verification stops. At this point, JMSeq checks if an implementation of the `VerificationFailureHandler` interface is present (cf. Section 3). If present, JMSeq tries to instantiate an instance of the verification failure handler and hands over the execution metadata to this handler. Otherwise, JMSeq terminates and reports how the specification is violated.

An event may also be sent due to occurrence of an exception. In this case, the execution is verified based on the exception specifications expressed using the mechanism described in Section 3. If the exception is allowed and verified the execution continues; otherwise, it fails and control is delegated to verification failure handler as explained above.

The execution verification component is composed of the following elements:

1. *Call expression* is a simple component for interacting with each state machine object. Basically, it transforms execution events coming from the JVM into call expressions used by the state machine components.
2. *Call expression state machine*: Sequences of executions are translated to "call expressions" that are to be accepted by a state machine. The state machine needs to distinguish the method's context on different calls to the same method. At

the end of a successful sequential execution, the state machine associated to that sequenced execution specification should be in a success state.

3. *Call expression builder* constructs a call expression out of:
   (a) A string which is in the format of the JMSeq specification grammar as in Fig. 3. This service is used the first time a sequential execution is detected to build the root of the future possible (candidate) call expressions.
   (b) An execution event; every time an execution is handed over to the verification module, an equivalent call expression is built for it so that it can be compared and matched against the call expression for the previous event.
4. *Call expression proposer* is a service proposing *possible next call expressions* for a given call expression based on the sequences specified in the annotation. As described by the grammar in Fig. 3, for each current call expression there can be several possible next call expressions that may appear as the next event, but for each of them the state machine associated with the grammar can only make one transition (that is, the specification is deterministic). Note that since more specification sequences are possible involving the same method, only those call expressions that are valid in all specifications will be proposed.
5. *Call expression matcher* is another service that tries to match two call expressions. It is used, for example, to validate the current call expression against all those proposed by the previous service. If a match is found, the execution continues; otherwise the verification is regarded as failed. In this case, if a verification failure handler is provided, the failure data is transferred to it for further processing.

5.2. JUnit support

To ease the process of verification, JMSeq provides a seamless integration to JUnit [4]. JUnit is a widely accepted unit testing framework for Java programs.

To use JUnit, the programmer is expected to write Java class files that introduce "test" methods. A test method is annotated with `@Test` annotation in JUnit. Each test method is run in isolation and independently of the other test methods. The programmer may provide special methods `setUp` and `tearDown` to, respectively, prepare resources or release them for the unit test. JUnit provides a runner framework that runs all Java classes containing test methods. Different tools such as Eclipse [5] or Maven [6] provide easy ways to run unit tests with JUnit.

JMSeq extends the JUnit framework and provides a simple to use mechanism for programmers to write unit tests in order to verify their programs. JMSeq introduces the `JUnit4Support` class that is to be extended by the programmer to direct the verification process. In this unit test class, the programmer uses the `@Test` annotation to identify the methods implementing a test, in the same way as in JUnit. Furthermore, the programmer should override the following methods such that they provide the information that JMSeq requires to start the verification process:

`getClassName()` *should* be overridden to provide the simple name of the class that contains the entry point to the application.

`getPackageBase()` *should* be overridden to provide the qualified package address that contains the class file provided by `getClassName()`.

`getExcludedPatterns()` is a method that may be overridden to provide the patterns that are excluded in matching the events received from the underlying virtual machine started by JMSeq. By default, a set of events received from the standard packages in Java are excluded. The developer may add, remove or modify the patterns based on the expected events.

`setUp()` is a standard base method from JUnit that is executed before each test method execution. It is supposed to prepare the pre-conditions before the actual test. If this method is overridden, it either has to call the parent method first or guarantee that hierarchy objects are ready for execution.

`tearDown()` is another standard base method that is executed after each test method.

`@Test` **methods** Each test method should carry `@Test` JUnit annotation to be considered a unit test method. Each test method should contain the call to `startJMSeq()`. Other than that, it is the choice of the tester what to add in the test method.

We describe how to use the JUnit support through an example. A `Basket` is a resource holder for apples. Generally, each `Basket` supports two basic features: `put()` and `get()`. The first one inserts an apple (i.e., an instance of a resource) to the basket while the second one retrieves and removes one apple from the basket. There are different resource producers and consumers: producers populate the resource holder (basket) and consumers use the resources available in the resource holder. As an example, `Pat` is a consumer and for now we forget about the producer that co-relates to the mutual `Basket` and `Pat`. `Pat` consumes from the `Basket` and gets apples to `eat()`.

In terms of JMSeq sequenced specifications, each invocation of `eat()` in `Pat` should be followed by an invocation of `get()` in `Basket`. We specify this scenario in Listing 5.

We need now to run the tests for the `Apples` application. This is made very easy by JMSeq's support for JUnit. We develop a JUnit test class for this purpose and provide a set of predefined information to the base test class. The JMSeq test classes should *extend* the `JUnit4Support` that is provided in the library. The code is presented in Listing 6.

Listing 5 Apples JMSeq specification.

```
@SequencedObject
public class Apples {

   private Pat pat;

   public Apples(Basket basket) {
      this.pat = new Pat(basket);
   }

   @SequencedMethod("{call(public void nl..Pat.eat())" +
         "[{call(public int nl..Basket.get())}$]}$")
   public void startLunch() {
     // generally a sequence of "eat" invocations
     pat.eat();
     // still hungry?
     pat.eat();
   }

   public static void main(String[] args) {
     Basket b = new Basket(10);
     Apples apples = new Apples(b);
     apples.startLunch();
   }
}
```

Listing 6 Apples JMSeq JUnit test unit.

```
public class ApplesDriver extends JUnit4Support {

   @Test
   public void startLunch() {
     startJMSeq();
   }

   @Override
   protected String getClassName() {
     return Apples.class.getSimpleName();
   }

   @Override
   protected String getPackageBase() {
     return Apples.class.getPackage().getName();
   }
}
```

6. The Fredhopper access server: a case study

Fredhopper[1] is a search, merchandising and personalization solution provider, whose products are uniquely tailored to the needs of online businesses. Fredhopper operates behind the scenes of more than 100 of the largest online shops.[2] Fredhopper offers its products and facilities to e-Commerce companies (referred to as customers) as services over the cloud computing infrastructure. In this respect, Fredhopper should tackle various challenges in resource management techniques, the customer cost model, and service level agreements. One of the major components of their software is the Fredhopper Access Server (FAS), which provides access to high quality product catalogs.

To orchestrate different services such as FAS or data processing, Fredhopper takes advantage of a service controller (referred to as Controller). The Controller manages different service installations for each customer. For instance, a customer can submit their data along with a processing request to their data hub server. The Controller then picks up this data and

[1] http://www.sdl.com/products/fredhopper/.
[2] http://www.sdl.com/campaign/wcm/gartner-maqic-quadrant-wcm-2013.html?campaignid=70160000000fSXu.

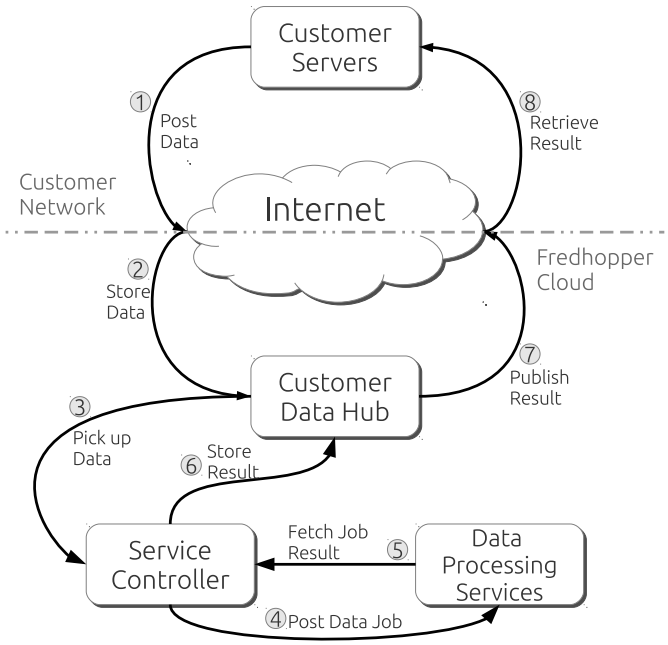

Fig. 6. Fredhopper's Controller life cycle for data processing requests from customers.

initiates a data processing job, usually in the cloud. When the data processing is complete, the result is published to the customer environment and becomes also available through FAS services. Fig. 6 illustrates this example scenario.

The Controller provides a simple client API for customers to develop applications that use Fredhopper services. Essentially a data job request consists of three major steps:

1. Prepare the data, upload it to the customer data hub and receive a data identifier.
2. Specify the data processing: reindexing, updating or loading data. Such actions are called *triggers*.
3. Collect the results of a trigger and decide how to use them.

The customer is required to have the appropriate data identifier before executing a new trigger. In fact, the customer application is supposed to use the proper sequence of operations from the API.

In this case study, we examined and measured how JMSeq can help customers improve their usage of client API for data processing; i.e. verify data processing triggers and notify customers as soon as they occur. We used "rebuild" and "update" triggers in this experiment based on customer environment data from the period of February 2013.[3] In this experiment, we measured the following:

1. Using the log files, we replayed the triggers twice with different settings.
2. In one setting, we replayed the original customer data processing trigger.
3. In the other setting, we replayed the original customer data processing that was wrapped with JMSeq verification.
4. In each setting, we measured the number of *successful* and *failed* triggers. A successful trigger is one for which data processing completes and the result is reported back to customer data environment.

Fig. 7 presents the measurements for each setting without using JMSeq and with using JMSeq for both rebuild and update triggers. The measurements were based on an estimated number of 30 300 triggers in the same period. Each figure shows the influence of using JMSeq for the data processing task. On the left, measurements shows that JMSeq verification for rebuilding the catalog helps to decrease failures by 15% if using JMSeq verification. On the right, measurements shows that JMSeq verification has less influence to improve update process as it naturally involves less erroneous data because of smaller data size in update triggers.

Next we present the case study in code listings in which JMSeq monitors calls to methods of the Controller API. The runtime environment stops as soon as an incorrect sequence of calls is detected.

The relevant part of the API of the Fredhopper Controller is presented in Listing 7. Customer applications use the interface to create a data object reference (using the `create(Object)` method) and then use the created data reference to start a

[3] Fredhopper data in this research is confidential due to privacy protection of customers.

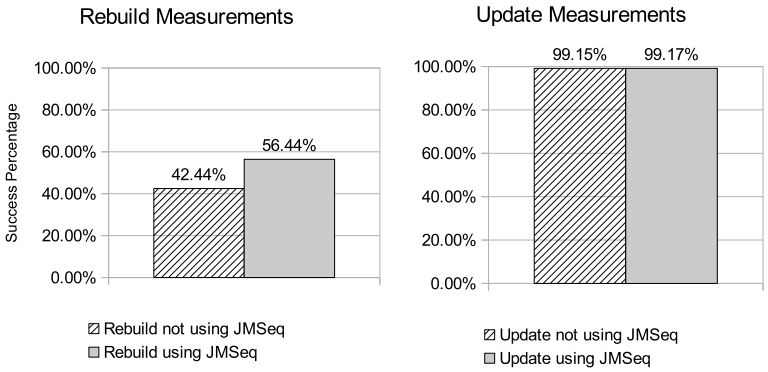

Fig. 7. Result of experiment measurements without and with using JMSeq on Fredhopper Client API directed with live environment data from customers.

Listing 7 Controller data dispatcher interface API.

```
1 interface DataDispatcher {
2    Data create(Object object);
3    Future<?> dispatch(String dataId);
4    Future<?> dispatch(Data oid, String dataId);
5 }
```

Listing 8 A use case with no data requirement.

```
1 class CustomerApp {
2    public void rebuildCatalog() {
3       Future<?> result =
4          dispatcher.dispatch("reindex");
5       // further processing of result
6    }
7 }
```

Listing 9 A use case with data requirement.

```
1 class CustomerApp {
2    public void updateCatalog(Catalog catalog) {
3       Data data = dispatcher.create(catalog);
4       Future<?> result =
5          dispatcher.dispatch(data.getId(), "update");
6       // further processing of result
7    }
8 }
```

data job (using the overloaded `dispatch` method, which has an optional `Data` parameter). Note that the dispatch method allows for two different uses by the customer, for instance in Listing 8 the data job to be executed performs only a rebuild of the customer's product catalog and as such does not need to have a data reference.

If the customer needs to update their catalog with new products, then first a data reference is created. This reference is used to initiate a data job, as described in Listing 9.

Note that we cannot expect to have the source code of the above listings. To verify the correct use of the Fredhopper Controller API with JMSeq we can create a `CustomerAppDriver` class which calls the `CustomerApp` methods and monitors the subsequent method calls. The expected usage of the API of the two scenarios above is described by the SequencedMethod annotations in Listing 10. If the `CustomerApp` does not behave as expected, the verifier detects an error.

6.1. Discussion

We use JavaMOP [15,14,7] for the same case study from Fredhopper. Listing 11 shows a JavaMOP script to describe the same verification goal.

Listing 10 How to verify Fredhopper API usage.

```
@SequencedObject
class CustomerAppDriver {

   @SequencedMethod(
      "{call(* nl..CustomerApp.rebuildCatalog()" +
      "[{call(* nl..DataDispatcher.dispatch(String))}]}")
   public void verifyRebuildCatalog() {
      // initiate the customer application
      app.rebuildCatalog();
      // finalize verification
   }

   @SequencedMethod(
      "{call(* nl..CustomerApp.updateCatalog(Catalog)" +
      "[{call(* nl..DataDispatcher.create(Object)}" +
      "{call(* nl..DataDispatcher.dispatch(Data, String))}]}")
   public void verifyUpdateCatalog(Catalog catalog) {
      // initiate the customer application
      app.updateCatalog(catalog);
      // finalize verification
   }
}
```

Listing 11 Fredhopper case study with JavaMOP.

```
UpdateCatalog(Catalog catalog) {
   event create_data before(DataDispatcher d):
      call(* DataDispatcher.create(..)) && target(d) {}

   event dispatch_data before(DataDispatcher d):
      call(* DataDispatcher.dispatch(..)) && target(d) &&
      cflow(dispatch_data) {}

   ere: (create_data dispatch_data)

   @fail { __RESET; }
}
```

JavaMOP uses aspect-orientation and Java bytecode instrumentation. Using aspects (e.g., as in AspectJ [20]) means that the eventual customer application bytecode is instrumented. The question is *how can it be proved that JavaMOP does not have any influence on the application?* If the customer is not willing to cooperate in such a process, the system cannot be monitored with JavaMOP. JMSeq on the other hand does not change the source- or bytecode, because JMSeq runs on a different JVM as the one running the customer application.

7. Performance results

In this section, we present performance measurements of JMSeq and other techniques to verify a program. We choose a problem that has been studied in the context of JavaMOP [15] to provide a reference for comparison.

The example verifies the correct use of `java.util.Iterator`. Originally, the interface introduces two major methods: `hasNext()` and `next()`. The first one checks whether there are still elements in the sequence that can be visited and the second one gives out the next element in the sequence and moves forward. An invocation of `next()` presumes that it succeeds an invocation of `hasNext()`. Otherwise, an invocation of `next()` may lead to an exception.

We use JMSeq to verify an example code for this scenario. We use the code snippet displayed in Listing 12 that uses JMSeq annotations to express the correct usage. In the code, we use a variable `count` to control the number of times `hasNext()` is called before there is a call to `next()`. This example always passes the JMSeq verification with success. For performance measurements, we study the "correct" use of the example while verification of wrong usages are also possible.

In the following, we provide certain measurements from executions of JMSeq for the example above for different values of `count` in {10 000, 15 000, 20 000, 30 000, 50 000, 100 000, 200 000}. We use VisualVM tool that is a standard profiling and performance measurement tool packaged with standard Java development packages.

Listing 12 Iterator usage with JMSeq annotations.

```
1   @SequencedMethod(
2     value = "{call(* java..Vector.iterator())}$"
3           + "{call(* java..Iterator.hasNext())}$"
4           + "{call(* java..Iterator.next())}",
5         allowExceptions = false)
6   public void visitAllElements() {
7       Iterator<Integer> i = v.iterator();
8       for (int k = 0; k < count && i.hasNext(); ++k) {
9           i.hasNext();
10          i.next();
11      }
12  }
```

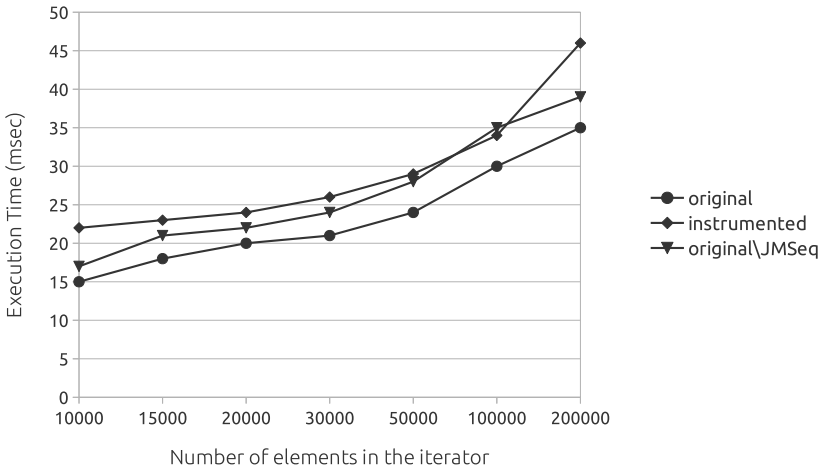

Fig. 8. Comparison of different techniques applied to the same program. In this figure, "original" denotes the original program that is measured for execution time; "instrumented" denotes using a technique such as JavaMOP that uses instrumentation for verification; and "original/JMSeq" denotes the measurement of the execution time for the program apart from times for JMSeq.

We measure the execution time of the following cases:

Original. We execute the original program and measure the execution time of the program with the above `count` values.
Instrumented. We use a technique that uses instrumentation such as JavaMOP [15] and apply it to the program and measure the execution time the same.
Original/JMSeq. We execute the original program and monitor it by JMSeq. We measure the execution time of the original program while being verified by JMSeq.

Fig. 8 displays how different cases compare to each other regarding the execution time. As expected, the original program execution has the lowest execution time. Since techniques based on instrumentation modify the bytecode of the program, this adds to the execution time of the program. The figure shows that compared to instrumentation techniques, JMSeq adds a lower overhead.

We use the performance results here to argue that the technique used by JMSeq is more suitable to verify applications that are mission-critical or operate on a high load. Experience shows that, under high load, issues start to emerge that do not exist under normal operational situations. In such circumstances, the application provider is very likely to look for methods to verify and detect problems. And yet, they need to make sure that the application remains at least at the same level of execution and performance. Moreover, the fact that JMSeq runs in a separate JVM increases the safety of our approach, since JMSeq can be used from even a remote machine to verify a program, without the slightest influence on the resources that run the target program.

8. Related work

JML [23] provides a robust and rigorous grounds for specification of behavioral checks on methods. Although JML covers a wide range of concerns in assertion checking, it does *not* directly address the problem of method call sequence specification

as it is more directed towards the reasoning about the state of an object. JML is rather a comprehensive modeling language that provides many improvements to extensions of Design by Contract (DBC) [25] including jContractor [8] and Jass [12].

In [16], taking advantage of the concept of *protocols*, an extension to JML is proposed that provides a syntax for method call sequences (protocols) along with JML's functional specification features. Through this extension, the developer can specify the methods' call sequence through a *call sequence clause* in JML-style meta-code. In the proposed method, the state of a program is modeled as a "history" of method calls and return calls using the expressiveness of regular expressions; thus, a program execution is a set of "transitions" on method call histories. The verification takes place when the execution history is simulated using a finite state machine and checked upon the specified method call sequence clause.

JML features have been equivalently implemented with AspectJ constructs [27], using aspect-oriented programming [20]. The authors of this work propose AJMLC (AspectJ JMLC), which integrates AJC and JMLC into a single compiler so that instead of JML-style meta-code specifications, the developer writes *aspects* to specify the requirements.

In [18] an elegant extension of JML with histories is presented. Attribute Grammars [22] are used as a formal modeling language, where the context-free grammar describes properties of the control-flow and the attributes describe properties of the data-flow, providing a powerful separation of concerns. A runtime assertion checker is implemented. Our approach differs in several respects. The implementation of their runtime checker is based on code instrumentation. Additionally, they use local histories of objects, so call-backs cannot be modeled. However, the behavior of a stack can be modeled in their approach (and not in ours), since their specifications are not regular expressions but context-free languages.

In the domain of runtime verification, Tracematches [9] enables the programmer to specify events in the execution trace of a program that could be specified with "the expressiveness" of a regular pattern. The specification is done with AspectJ pointcuts and upon a match the advised code is run for the pointcut. Along the same line, J-LO [13] is a tool that provides temporal assertions in runtime-checking. J-LO shares similar principles as Tracematches with differences in specifications using linear time temporal logic syntax.

Additionally, Martin et al. propose PQL [24] as a program execution trace query language. It enables the programmer to express queries on the execution events of objects, methods and their parameters. PQL then takes advantage of two "static" and "dynamic" checkers to analyze the application. The dynamic checker instruments the original code to install points of "recovery" and "verification" actions. The dynamic checker also translates the queries into state machine for matching criteria. The set of events PQL can deal with includes method calls and returns, object creations and end of program among others. Accordingly, generic logic-based runtime verification frameworks are proposed as in MaC [21], Eagle [11], and PaX (PathExplorer) [19] in which monitors are instrumented using the specification based on the language specific implementations.

Using runtime verification concepts, Chen and Roşu propose MOP [14,7] as a generic runtime framework to verify programs based on monitoring-oriented programming. As an implementation of MOP, JavaMOP [15] provides a platform supporting a large part of runtime JML features. Safety properties of a program are specified and inserted into the program with monitors for runtime verification. Basically, the runtime monitoring process in MOP is divided into two orthogonal mechanisms: "observation" and "verification". The former stores the desired events specified in the program and the latter handles the actions that are registered for the extracted events. Our approach follows the same idea as MOP, but it does not use AOP to implement it. Another major difference is in the specifications part. MOP specifications are generic in four orthogonal segments: logic, scope, running mode and event handlers. Very briefly, the *scope* section is the fundamental one that defines and specifies the parts of the program under test. It also enables the user to define desired events of the program that need to be verified. The *logic* section helps the user specify the behavioral specification of the events using different notations such as regular expressions or context-free grammars. The *running mode* part lets the user specify what is the running context of the program under test; for instance, if the test needs to be run per thread or in a synchronized way. And, *the event handlers* section is the one to inject customized code of verification or logic when there is a match or fail based on the event expression logic. In Section 6.1, we discussed how JMSeq contrasts with JavaMOP and similar approaches.

9. Conclusion and future work

We proposed JMSeq, a framework for specifying sequences of possibly nested method calls using Java annotations. The sequences do not only consist of method names, but may contain information such as object caller and callee, and distinguish overloaded methods. JMSeq uses Java Platform Debugger to monitor the execution of a component-based system in Java. Monitoring is divided into two phases: observing the events in the program and verifying them at runtime against the local specifications provided through annotated methods. JMSeq does not use *any* form of instrumentation; neither source level nor bytecode level. Moreover, JMSeq provides a simple way to express exceptions and verify them in the execution of the target application. JMSeq is integrated with JUnit. We presented a case study from Fredhopper distributed cloud services in the domain of e-commerce and marketing. Moreover, we provided a set of performance results for JMSeq in different settings for program execution and runtime verification.

JMSeq is a novel approach to runtime verification of software using code annotations. The approach is especially suitable for runtime component-based verification. We improved the execution data model of JMSeq using optimization techniques such as in-memory object caches with indices. In the context of event processing, we examined different implementations

of JPDA and JDI standards (e.g. using Eclipse platform [1]) and it showed part of the optimization is out of our reach. We used VisualVM, a standard Java profiling tool, to measure and discuss the performance of JMSeq.

A future line of work consists of porting JMSeq to multi-core platforms. Concurrent and distributed software introduces new challenges for runtime verification of applications. In a concurrent setting, objects interact with each other from different threads of execution (which may optionally involve synchronization and locking events). JMSeq should be able to verify such objects in different threads and consider how synchronization and locking may affect the runtime or flow of an application. In a distributed context, application objects are located in (physically) remote JVM instances. In such scenarios, JMSeq needs to load or observe a topology of the involved JVMs and their objects to be able to verify the application for correct sequences of method calls.

We plan to extend JMSeq by providing native features such as mock [17] implementations in case parts of the system are unavailable. JMSeq may provide better integration for mocking parts of the system that are not implemented or not available. Such features will be especially useful in the context of integration testing and verification. At the time of integration, not all components of the system may be available and complete; and yet, the application should be verified in the parts that are complete. In comparison with runtime checking, another line of future work can be to extend JMSeq to support static verification of protocols (using approaches as in Copilot [26]).

Finally, we plan to extend the specification language to a context free language. This is in fact simple as JMSeq already implements a simple form of pushdown automata to recognize the call-return structure of Java. In fact, Java is expressible by a restricted form of deterministic pushdown automata, called visibly pushdown automata [10], where push and pop correspond to call and return, respectively.

References

[1] Eclipse debug platform, http://www.eclipse.org/eclipse/debug/.
[2] Java 5 annotations, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.
[3] JPDA reference home page, http://java.sun.com/javase/technologies/core/toolsapis/jpda/.
[4] JUnit test framework, http://www.junit.org/.
[5] JUnit tutorial in eclipse, http://www.vogella.com/articles/JUnit/article.html.
[6] Maven JUnit integration with surefire plugin, http://maven.apache.org/surefire/maven-surefire-plugin/examples/junit.html.
[7] MOP: Monitoring-oriented programming, http://fsl.cs.uiuc.edu/index.php/MOP.
[8] P. Abercrombie, M. Karaorman, jContractor: Bytecode instrumentation techniques for implementing design by contract in Java, Electron. Notes Theor. Comput. Sci. 70 (4) (2002) 55–79.
[9] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, 2005, pp. 345–364.
[10] R. Alur, P. Madhusudan, Adding nesting structure to words, J. ACM 56 (2009) 16:1–16:43.
[11] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, in: B. Steffen, G. Levi (Eds.), Verification, Model Checking, and Abstract Interpretation, 5th International Conference (VMCAI'04), Proceedings, in: Lect. Notes Comput. Sci., vol. 2937, Springer, 2004, pp. 44–57.
[12] D. Bartetzko, C. Fischer, M. Möller, H. Wehrheim, Jass – Java with assertions, Electron. Notes Theor. Comput. Sci. 55 (2) (2001) 103–117.
[13] E. Bodden, J-LO, a tool for runtime-checking temporal assertions, Master thesis, RWTH Aachen University, 2005.
[14] F. Chen, G. Roşu, MOP: an efficient and generic runtime verification framework, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA '07, 2007, pp. 569–588.
[15] F. Chen, G. Rosu, Java-mop: A monitoring oriented programming environment for Java, in: N. Halbwachs, L.D. Zuck (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, Proceedings, in: Lect. Notes Comput. Sci., vol. 3440, Springer, 2005, pp. 546–550.
[16] Y. Cheon, A. Perumandla, Specifying and checking method call sequences of Java programs, Softw. Qual. J. 15 (March 2007) 7–25.
[17] F.S. de Boer, M.B. Bonsangue, A. Gruner, M. Steffen, Java test driver generation from object-oriented interaction traces, Electron. Notes Theor. Comput. Sci. 243 (0) (2009) 33–47.
[18] S. de Gouw, J. Vinju, F.S. de Boer, Prototyping a tool environment for run-time assertion checking in JML with Communication Histories, in: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTFJP'10, ACM, 2010, pp. 6:1–6:7, http://dx.doi.org/10.1145/1924520.1924526.
[19] K. Havelund, G. Rosu, Monitoring Java programs with Java PathExplorer, in: Runtime Verification (in connection with CAV '01), Electron. Notes Theor. Comput. Sci. 55 (2) (2001) 200–217.
[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting started with ASPECTJ, Commun. ACM 44 (October 2001) 59–65.
[21] M. Kim, M. Viswanathan, S. Kannan, I. Lee, O. Sokolsky, Java-MaC: A run-time assurance approach for Java programs, Form. Methods Syst. Des. 24 (March 2004) 129–155.
[22] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 2 (2) (1968) 127–145.
[23] G.T. Leavens, A.L. Baker, C. Ruby, Preliminary design of JML: a behavioral interface specification language for Java, Softw. Eng. Notes 31 (May 2006) 1–38.
[24] M. Martin, B. Livshits, M.S. Lam, Finding application errors and security flaws using PQL: a program query language, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, 2005, pp. 365–383.
[25] B. Meyer, Object-Oriented Software Construction, 2nd ed., Prentice Hall, New Jersey, 2000.
[26] L. Pike, Copilot: A stream language for generating hard real-time C code, http://leepike.github.com/Copilot/.
[27] H. Rebêlo, R.M.F. Lima, M. Cornélio, S. Soares, A JML compiler based on AspectJ, in: Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST'08), IEEE Computer Society, 2008, pp. 541–544.
[28] J. Shore, Fail fast, IEEE Softw. 21 (5) (2004) 21–25.