View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Monitoring Method Call Sequences using Annotations*

B. Nobakht[1], M.M. Bonsangue[1,2], F.S. de Boer[1,2], and S. de Gouw[1,2]

[1] Leiden Institute of Advanced Computing,
Universiteit Leiden, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
{bnobakht,marcello}@liacs.nl

[2] Centrum Wiskunde and Informatica,
Science Park 123, 1098 XG Amsterdam, The Netherlands
{frb,cdegouw}@cwi.nl

Abstract. In this paper we introduce JMSeq, a Java-based tool for the specification and runtime verification via monitoring of sequences of possibly nested method calls. JMSeq provides a simple but expressive way to specify the sequential execution of a Java program using code annotations via user-given sequences of methods calls. Similar to many monitoring-oriented environments, verification in JMSeq is done at run-time, but differently from all other approaches based on aspect-oriented programming, JMSeq does not use code instrumentation, and therefore is suitable for component-based software verification.

Keywords: Object monitoring, run-time verification, method call sequence specification, code annotation, component-based testing, communication traces

1 Introduction

Testing and, more recently, monitoring are two established approaches to the verification of large complex systems. Testing is generally used to validate the results of each step in the software life cycle against the expected ones. More recently, monitoring-oriented programming has emerged as a formal branch of testing suitable to validate runtime data collected during system execution [13]. While in ordinary testing the software system under test must be stimulated so to reproduce an expected behavior, in monitoring the actual behavior is observed and analyzed a posteriori with respect to some specification.

Component based software engineering advocates the construction of software by gluing together prefabricated components [27]. Because the code of the components is not always available, automatic code insertion is not possible, thus posing new challenges to testing and monitoring of the final product. An

* 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/)

alternative to the white box view of monitoring is given by automatic generation of wrappers for monitoring the external behavior of third-party components. A wrapper is automatically generated for every component with tracking code to form an observable component in the black box view. As for the automatic code insertion, formal specifications are separated from source code, but the complexity of generating the wrappers can be very high, as the code relative to a single specification must be distributed among several wrappers. The approach is thus not flexible and easily reusable. Basically, there are no other systematic methods and technologies available to control and monitor the external behaviors of the components according to a test specification.

In this paper we consider components as compiled Java packages that are annotated with specification of the internal object behaviors by means of sequences of outgoing method calls from the component and incoming return messages to the component. Component designers can use the standard Java 5 Annotations [3] to specify the intended behavior of the component as well as the action to be taken in case of failure. We express the intended behavior through a set of execution traces[3]. To check conformance of the execution trace with respect to its specification, monitors are automatically synthesized from specified properties and integrated into the original system to check its dynamic behaviors during execution. The approach is therefore simple, modular and flexible to use. It does not assume that the component source code is available, but it requires the addition of annotations either at design process or the testing phases, similar to what happens in the proof-carrying code approach [25]. The runtime verification and monitoring framework we implement is based on the Java Platform Debugger API (JPDA) [4]. Our method is thus complementing the approach taken in JavaMOP [14], where Java programs are monitored and verified at runtime by means of code instrumentation. JMSeq can be used to complement other existing testing and verification frameworks to add capabilities for the developer. For instance, one can use JML [22] to specify the properties of the data flow and use JMSeq to specify the properties of the control flow.

The rest of the paper is organized as follows. In Section 2 we discuss the problem of monitoring and testing component based systems. Then, in Section 3 we present a language for specifying sequences of method calls that are annotating the interfaces or the code of a program, as explained in Section 4. These annotations are used by the JMSeq framework, introduced in Section 5, to monitor the program. In Section 6, we discuss related work including a more detailed comparison with JavaMOP [14], another monitoring framework. Finally, we conclude in Section 7 and also discuss possible future work.

2 Monitoring component based systems

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

[3] By execution traces, we mean the sequence of method calls and method returns, not sequences of states

by the system specification. The observable information of the monitored system typically includes behaviors, 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. In the case that an execution violates 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 receive informative messages about the execution of an application at run time. New code is inserted into the original code, preserving the original logic of the application; yet, the extra code is essential for the management of monitoring and verification mechanisms. For example JavaMOP [14] uses AspectJ to inject monitors into the original code. Also, the JML run-time assertion checker has been implemented using aspect-oriented programming [26].

As programs have become larger and more complex, encapsulation and hiding techniques have become more important. Component based software construction has emerged as a viable solution for handling the complexity of software. Components implement one or more interfaces describing the services they provide and require. Usually, the interfaces are described only in terms of a signature and the source code of a component is not available. During the integration phase, when components are composed into a system, it is thus very difficult for the developer to check if the behavior of the system conforms to its specification. In fact, the absence of source code implies that we cannot instrument it, and therefore current run-time verification techniques cannot be used.

In this paper we present JMSeq, a runtime verification framework that is not based on code instrumentation, but rather on *code annotation*. Code annotation has become increasingly popular in the past few years, especially because of its effectiveness in integrating formal techniques for verification with programming. Essentially, annotations are code segments that are compiled but do not provide any logic or business in the program; yet, they indirectly affect the program execution based on the additional information they add to the running code.

Annotations are different from documentation tags as originally used by JML to specify assertions for verification of Java code [22]. In fact documentation tags are not compiled, and thus are highly dependent on the presence of the source code. The success of modern testing frameworks such as JUnit [5] advocated for development of services based on annotations for the recent versions of the Java language. JMSeq uses Java annotations to provide a way for the programmer to specify for each method a set of execution traces representing an abstraction of the intended behavior, or protocol, of a component. Thus, method call sequence specifications specify properties of the control flow of the global execution trace. The methods in these sequences denote the use of services of other components or even of the component itself. In particular we do not exclude call-backs.

When deployed, we assume that each component contains the description of its protocol that can be checked for conformance at run-time. Only the component interfaces need to be known to the system developer, possibly with JMSeq annotations when the source code is not available. Of course it would be advantageous if the developers have documentation describing the internal components annotations. The advantage of this approach is an easy integration of JMSeq with the testing framework JUnit [5] for the execution of an individual component in a specific context to see whether they generate the expected results. In this case, we do not need the component to be annotated at all as one can easily write a JUnit test for it, annotate JUnit methods with JMSeq annotations and run the JUnit test with JMSeq.

Let us consider the notions of "black-box" and "white-box" testing. As white-box testing usually utilizes the internal structure of a software system, the source code dependence is inevitable. However, in the case of black-box testing, there is no need for the source code of the system as the test depends on the process, input and output, and the test specification of the system.

Moreover, in the field of testing, there are times that some parts of the system are not available (or made unavailable); thus, there should be a mechanism to substitute "mock implementation" for a system interface when needed. This technique may be referred to as *unit testing* in contrast with *integration testing* as different components are being regarded as stand-alone entities providing predefined interface and behavior and no mock implementation can be substituted or provided in the system. In other words, in integration testing, the system may be complete and operational on its own and in unit testing, there are points that require mock implementations (since the development is not complete yet).

JMSeq is uses black-box testing as it does not use the internal structure of the program. And, it supports both unit testing and integration testing techniques; provided that the developer or the tester provides the mock implementations required in unit testing.

Table 1 characterizes several testing frameworks by distinguishing among code annotation, code instrumentation, unit testing, and integration testing techniques.

	Instrumentation	Annotation	Technique
JML	✓	×	Unit, Integration
JavaMOP	✓	×	Unit, Integration
PQL	✓	×	Unit, Integration
JMSeq	×	✓	Unit, Integration

Table 1: Approach Comparison

3 Method Sequence Call 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 between a small, fixed number of objects. In Figure 1, we see two UML message sequence diagrams each describing how the four objects interact with other and in which order. In this section, we develop a specification language for describing kinds of interactions using a context free grammar.

Essentially, a specification language for sequences of method calls needs to distinguish between the specifications of the two cases in Figure 1.

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. For instance, 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*.

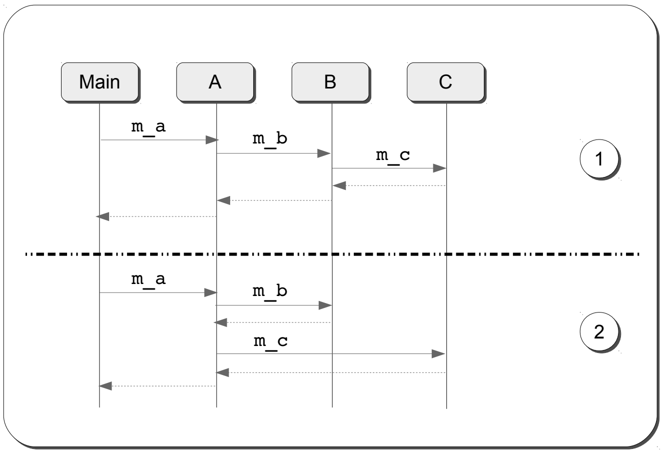

Fig. 1: Examples of Method Sequence Call Specification

It is clear from the above examples that regular expressions over method calls are not enough to specify the typical nested call structure of a sequence of messages in the presence of recursion. In general, such sequences form a context free language. However they have a special structure: there is a deterministic pushdown automaton accepting them such that it pushes or pops at most one symbol only, depending if a method call or return is read, respectively. Such an automaton is called a visibly pushdown automaton [8].

In JMSeq, a specification denotes 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 belong to a context free language specified by means of a grammar. Figure 2 represents the method sequence call specification grammar.

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

Fig. 2: Method Sequence Specification Grammar

A *specification* consists of a sequence of calls that can be repeated an a priori fixed number of times ("$\langle \textit{Specification} \rangle^{m}$"), with $m \geq 0$, one or more times ("$\langle \textit{Specification} \rangle \$$"), or zero or more times ("$\langle \textit{Specification} \rangle \#$"). Although JMSeq does not use aspect-oriented programming in its implementation; we have used the generic method call join points syntax of AspectJ [19], using for example # to denote the more standard Kleene star operation. Additionally, $(\langle \textit{Specification} \rangle)$ 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 time, zero or more times, or exactly m-times, for some $m \geq 0$. Additionally, the wild card $\{\texttt{call}(*)\}$ denotes a call to an arbitrary method. To support branching, JMSeq also provides $< \langle \textit{Call} \rangle ? \langle \textit{Call} \rangle >$ to allow the specification of a choice in the 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 by using parentheses. Information about the message call, like caller, callee, method name, and actual parameters are expressed using the generic AspectJ syntax for pointcut model that is used to express the point of calling a method from another object [19,1]. Put it simply, a call signature is of the form

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

reflecting the method declarations in Java that include method names, method parameters, return types, modifiers like "static" or "public", and throws clauses. It is noticeable that the annotation are used for the "public" method from outside the component; yet, the developer is allowed other modifiers such as "private" for runtime checking since the internally used annotations are not visible from outside the component. 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. Method modifiers and return type are specified by the two leftmost patterns. The method name is followed by a specification of the type of the parameters passed during the call, and possibly by a specification of the throws clause. 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 few example of correct method sequence specifications. For instance, the specification of the sequence in case 1 of Figure 1 is given by:

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

Here is important to notice that the call to method `m_b` is internal to `m_a`, and the one 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 m_a will satisfy a requirement in advance that is used by m_b or m_c.

It is *notable* that the meta-grammar provided in Figure 2 is a context-free grammar; however, the actual specifications are regular expressions; for example, they may contain unbounded repetition and choice of calls. They are not context free as the bound m in the repetition is assumed to be fixed and not a free variable.

4 Annotations with method sequence calls

Annotations do not directly affect 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 [3]. Annotations can be read from source files, class files, or reflectively at run-time. Once an annotation type is defined, it can be used to annotate declarations. An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as `public`, `static`, or `final`) can be used.

One of the major desired effects of using annotations in code is that it will allow for testing components without the need to have their source code. We only used the meta data loaded from the annotations during runtime.

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

Sequenced Object Annotations Simply put, `SequencedObject` annotation is just a marker for those classes to notify the annotation meta-data loader that the objects from the annotated class contain methods which specify a sequential execution. The code is demonstrated in Listing 1.

Listing 1: SequencedObject Annotation Declaration

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

`@Retention(RetentionPolicy.RUNTIME)` declares that this annotation is only applicable during runtime and may not be used in other scenarios, whereas `@Target(ElementType.TYPE)` declares that this annotation can only be used on types including classes, interfaces and enumerated types.

Sequenced Method Annotation A sequence method annotation is used to specify the sequence of method calls under a given method. The annotation requires a string property declaring the sequential specification discussed in Section 3. Listing 2 presents the declaration.

Listing 2: SequencedMethod Annotation Declaration

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface SequencedMethod {

  String value();

  Class<VerificationFailureHandler> verificationFailureHandler();
}
```

`@Target.ElementType.METHOD` declares that this annotation is only applicable to *methods*. The string value from `value()` holds the sequential specification. The class `VerificationFailureHandler` is introduced by `verificationFailureHandler()` and is used when a sequence execution failure occurs at runtime. Its implementation is left to the tester or developer who should provide a custom behavior to handle the verification failures.

In Listing 3 we give two examples of annotations of the class `Main` class in Figure 1. In both cases we annotated method `main()` with two sequences of method calls, describing the behaviors given in Figure 1

Listing 3: Sample annotated specification

```

// Case 1
@SequencedObject
public class Main {

  @SequencedMethod("{call(* *.A.m_a(..))[{call(* *.B.m_b(..))[{call(* *.C.m_c
       (..))}]}]}")
  public void main() {
    // ...
  }

  public void init() {
    // ...
  }
}

// Case 2
@SequencedObject
public class Main {

  @SequencedMethod("{call(* *.A.m_a(..))[{call(* *.B.m_b(..))}][{call(* *.C.
       m_c(..))}]}")
  public void main() {
    // ...
  }

  public void init() {
    // ...
  }
}
```

5 The JMSeq framework

In this section, we present the JMSeq monitoring and testing framework, and discuss its implementation that uses Java 5 Annotations [3] and Java Platform Debugger API [4]. More implementation details, including samples and documentation, can be found at `http://code.google.com/p/jmseq/`.

As discussed above, we assume that Java components come together with sequenced object and sequenced method annotations. More annotations are possible within the same class, but not for the same method. This implies a local and partial view of the component specification, that can be scattered among all its constituent classes. The annotated methods are the ones that will be monitored by JMSeq.

JMSeq, based on some initial parameters, initiates another *inner Java virtual machine (JVM)* inside the current execution to control the sequenced execution of the program (Figure 3). Essentially, the parameters tell the inner JVM what type of events are going to be reported back to JMSeq for verification such as method entries and method exits. Also, JMSeq is not interested in events from all objects but only for those specified in the parameters. The inner JVM takes advantage of the Java Platform Debugger Architecture (JPDA) to access the needed details on the execution of the system while it is running. JPDA is a framework for debugging and interfacing the JVM. Java Debugger Interface is the interface of JPDA that gives access to different details on the execution of a program.

Before execution, JMSeq inspects the classes in the running class path to collect and store all the methods that are annotated for sequenced verification. This step, denoted in Figure 3, does not need code inspection, but uses the annotated meta data available in the compiled code.

The program is now executed. Whenever an event is reported to JMSeq the event trace model is updated in such a way that events are aware of their previous call stack trace (Figure 3): if the event represents a method that does not need to be monitored the execution continues until the next event, otherwise it is verified if it is an expected event. The verification is done through a simple state machine with a stack for the nested events occurred so far. The verification process is described as (Figure 3):

1. A "call expression" of the event model is constructed.
2. Using the meta data available from annotations of the methods, then, the next "possible call expressions" of the current state is built.
3. A match making is done between the possible call expressions and the current call expression as the candidate. The possible call expressions are computed on-the-fly rather than constructing the full automaton in the beginning of the verification process. This way we avoid the construction of states in the automaton that are never used in the execution. To avoid repeated computations of the same states, JMSeq utilizes dynamic programming techniques. If a match is found, the method is accepted; otherwise it fails. When a failure occurs, JMSeq will execute the custom verification handler that is imple-

mented either internally as part of the component code by the programmer, or externally by the system designer.

The overall JMSeq process is depicted in Figure 3.

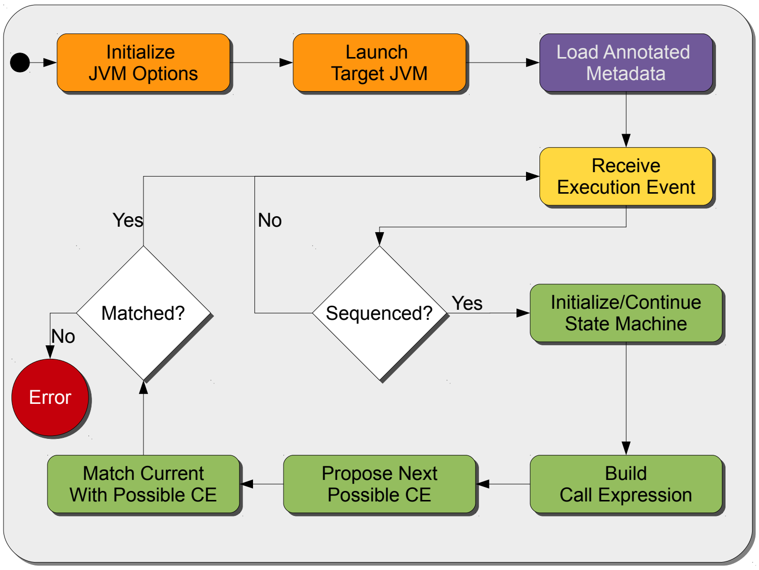

Fig. 3: Runtime Object Monitoring and Sequenced Execution Verification

5.1 JMSeq Architecture

The overall architecture of JMSeq is given in Figure 4. It basically consists of three main modules: one for handling the communication with the JVM executing the program, another module for storing the annotation information, and a third module for executing the run-time verification.

The current design of JMSeq is completely *general* and MODULAR; as it allows for replacing the grammar in Figure 2 with other specification modules, based, for example, on temporal logics or extended regular expressions.

Program Execution Trace Model and Processing According to 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.

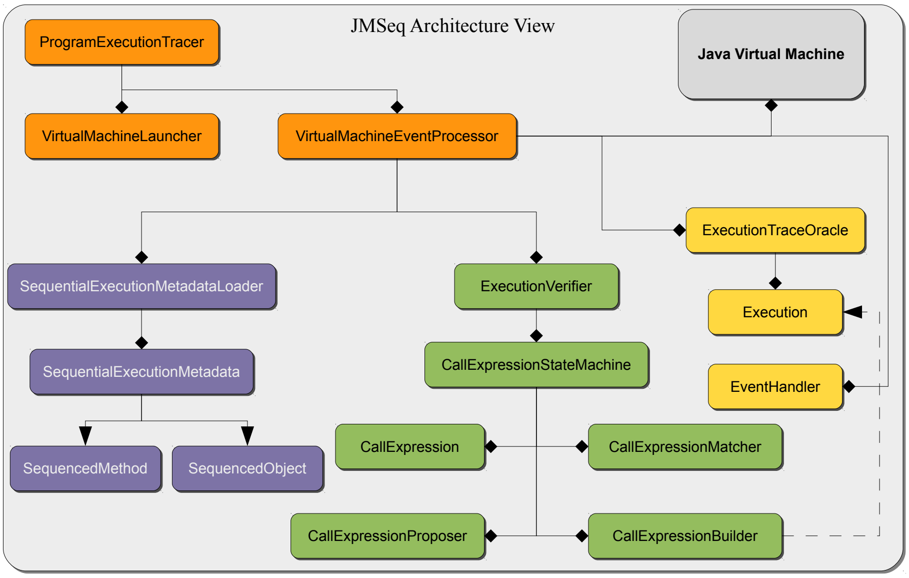

Fig. 4: Software Architecture for Method Sequence Specification

1. *Execution* is the central component that holds the information about every execution in the JVM using the JDI event mechanisms. Relevant events include “method entry” and “method exit”. 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 help traversing the executions at validation time or for the simulation of formal specification.
2. *EventHandler* is the event handling interface that is injected into JVM with access to JDI information. The event handler receives event for which it is subscribed and possibly takes an associate action. In particular it creates an instance of `Execution` for each sequence annotation and it stores it in the `ExecutionTraceOracle` registry ready to be used by the verification module. In general, all events received and processed by `EventHandler` are stored in this registry for further use by other components.

Sequential Execution Annotation Repository 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 sequential execution annotation repository to store this information. A service is responsible to read and load the meta-data of all classes that are annotated. This information is used when there is a need to verify the conformance of an execution event.

Execution Verification During an execution, events are received that need to be monitored and verified against a message sequence specification. For every specification in the meta-data repository, a deterministic state machine *including a stack* is created for recognizing the sequences in the specification. Whenever an event is processed that belongs to a sequence of method calls, the execution verifier checks if the state machine can execute a transition associated with this event. If the state machine accepts the current event, the execution will continue; otherwise, the execution is "invalid" and therefore it is stopped. At this point, JMSeq provides a simple way to plug in a verification failure handler by means of a class provided by either the component designer or the test developer. The verification failure handler should implement an interface introduced by JMSeq.

The execution verification component is composed by 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 expression"s that are to be accepted by a state machine. The stack of the automaton is necessary to distinguish the method's context on different calls to the same method calls, for example. At the end of a successful sequential execution, the stack for the automaton associated to that sequenced execution specification should be empty. Otherwise the top of the stack is used to match the event with a possible candidate call expression of the previous event.
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 Figure 2. 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 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 Figure 2, 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 automaton associated with the grammar can only make one transition (that is, the specification is a deterministic

context free language). 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 hander is provided, the failure data is transferred to it for further processing.

6 Related Work

JML [22] 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 sequence call 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 other extensions to Design by Contract (DBC) [24] equivalent formalism such as jContractor [9] and Jass [11].

In [16], taking advantage of the concept of *protocols*, an extension to JML is proposed that provides a syntax for method sequence calls (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 [26], using aspect-oriented programming [19] They propose AJMLC (AspectJ JMLC) that integrates AJC and JMLC into one compiler so that instead of JML-style meta-code specifications, the developer writes *aspects* to specify the requirements.

In [17] an elegant extension of JML with histories is presented. Attribute Grammars [21] 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 run-time assertion checker is implemented. Our approach differs in several respects. The implementation of their run-time checker is based on code instrumentation. Additionally, they use local histories of objects, so callbacks can not 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 [7] 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 [12] 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, Livshits and Lam propose PQL [23] 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 [20], Eagle [10], and PaX (PathExplorer) [18] in which monitors are instrumented using the specification based on the language specific implementations.

Using runtime verification concepts, Chen and Rosu propose MOP [15,6] as a generic runtime framework to verify programs based on *monitoring-oriented programming*. As an implementation of MOP, JavaMOP [14] 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 segment: 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 Listing 4 we show an example of JavaMOP code with ERE logic for the two specifications given in Figure 1.

Listing 4: Sample JavaMOP Specification using CFG logic

```
SampleCase1(Main m) {
  event method_m_a before(A a):
    call(* A.m_a(..)) && target(a) {}
  event method_m_b before(B b):
    call(* B.m_b(..)) && target(b) && cflow(SampleCase1_method_m_a) {}
  event method_m_c before(C c):
    call(* C.m_c(..)) && target(c) && cflow(SampleCase1_method_m_a) && cflow(
         SampleCase1_method_m_b) {}

  ere: (method_m_a method_m_b method_m_c)*

  @fail {
    System.err.println("Invalid Execution");
    __RESET;
  }
}

// Case 2
SampleCase2(Main m) {
  event method_m_a before(A a):
    call(* A.m_a(..)) && target(a) {}
  event method_m_b before(B b):
    call(* B.m_b(..)) && target(b) && cflow(SampleCase1_method_m_a) {}
  event method_m_c before(C c):
    call(* C.m_c(..)) && target(c) && cflow(SampleCase1_method_m_a) && !cflow(
         SampleCase1_method_m_b) {}

  ere: (method_m_a method_m_b method_m_c)*

  @fail {
    System.err.println("Invalid Execution");
    __RESET;
  }
}
```

It is interesting to note that both MOP specifications have *the same ERE expression*. This is because JavaMOP has separated logic and scope specification from event handling. The difference in monitoring is obtained by the different usage of the command `cflow` in the scope section. This command is an AspectJ construct to control the context of the execution when running some code inside a method [19].

7 Conclusion and future work

We proposed JMSeq, a framework for specifying sequences of method calls using Java annotations. The sequences do not only consist of method names, but may contain information such as object caller and callee. JMSeq uses Java Platform Debugger to monitor the execution of a component based system based on Java. Monitoring is divided into two phases: observing the events in the program and verifying them against the local specifications provided through annotated objects at runtime. No code instrumentation is necessary, as only binary code is enough for system testing and monitoring purpose. JMSeq can be integrated with JUnit for unit testing purposes. An initial version of JMSeq runtime checker is available at `http://code.google.com/p/jmseq`.

We believe JMSeq is a novel approach to runtime verification of software using code annotations. The approach is especially suitable for runtime component based verification, as it does not require the presence of source code. In line with this end, we plan to extend the support of JMSeq by providing native features such as mock implementation or symbolic execution in case parts of the system are unavailable which is particularly useful in unit testing. Currently, JMSeq only provides testing capabilities in the context of JUnit.

Another potential area of improvement is the data model used by JMSeq. Currently it overlaps with the event model that JPDA provides when publishing the registered events in JVM. The resulting overhead is rather expensive. Optimization will allow, for example, to store only the minimal necessary information providing a faster indexing for the retrieval of the events. Further performance improvement can be obtained by better exploiting the connections between JPDA and JVM. For example, there are tools such as Eclipse Debug Platform [2] that provide extensive facilities through JPDA with high performance. Currently, in JMSeq, it is the standard JVM that runs the program and publishes the events that are interesting to JMSeq for further verification for which, in turn, JMSeq uses a simple state machine to verify the executing events. As another future work, instead of using a state machine, one could use a pushdown automaton which allows for context-free method call sequence specifications. Moreover, in another approach, JVM, JPDA and the state machine can be merged together such that it is actually the JVM that asks permission for the next execution from the state machine that is provided. Thus, the testing framework can even take control of the underlying program execution for further checks or verification. In other words, method call sequence specifications may also be used in the pre-conditions of a method. In comparison with the runtime checking, another line of future work can be to extend JMSeq to support static verification of protocols.

Acknowledgments We thank the referees for their comments and Michiel Helvensteijn for the helpful discussion.

References

1. *AspectJ Language Semantics.* http://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html.
2. *Eclipse Debug Platform.* http://www.eclipse.org/eclipse/debug/.
3. *Java 5 Annotations.* http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.
4. *JPDA Reference Home Page.* http://java.sun.com/javase/technologies/core/toolsapis/jpda/.
5. *JUnit Test Framework.* http://www.junit.org/.
6. *MOP: Monitoring-oriented programming.* http://fsl.cs.uiuc.edu/index.php/MOP.
7. C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *OOPSLA*, 2005.
8. R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56:16:1–16:43, 2009.
9. P. Anercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing dbc in Java. *RV'02*, 2002.

10. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. *VMCI'04*, 2004.
11. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. *RV'01*, 2001.
12. E. Bodden. J-lo, a tool for runtime-checking temporal assertions. *Master Thesis, RWTH Aachen University*, 2005.
13. F. Chen and G. Rosu. Towards Monitoring-Oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, Oct 2003.
14. F. Chen and G. Rosu. Java-MOP: A monitoting oriented programming environment for Java. *TACAS*, 2005.
15. F. Chen and G. Rosu. MOP: An Efficient and Generic Runtime Verification Framework. *OOPSLA, ACM Press*, 2007.
16. Y. Cheon and A. Perummandla. Specifying and checking method call sequences of Java programs. *Springer Science*, 2007.
17. S. de Gouw, J. Vinju, and F. S. de Boer. Prototyping a tool environment for run-time assertion checking in JML with Communication Histories. *FTfJP'10*, 2010.
18. K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. *RV'01*, 2001.
19. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with ASPECTJ. *ACM CACM*, 2001.
20. M. Kim, S. Kannan, I. Lee, , and O. Sokolsky. Java-MaC: a Runtime Assurance Tool for Java. *RV'01*, 2001.
21. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
22. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering*, 2006.
23. M. Martin, V.B. Livshits, and M.S. Lam. Finding application erros and security flaws using PQL: a program query language. *OOPSLA*, 2005.
24. B. Meyer. *Object-Oriented Software Construction, 2nd edition.* Prentice Hall, New Jersey, 2000.
25. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.
26. H. Rebelo, S. Soares, R. Lima, P. Borba, and M. Cornelio. JML and Aspects: The benefits of instrumenting JML features with AspectJ. 2008.
27. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming.* Addison-Wesley, 2002.