JASSDA TRACE ASSERTIONS[*]

Runtime Checking the Dynamic of Java Programs

Mark Brörkens
University of Oldenburg
Mark.Broerkens@informatik.uni-oldenburg.de

Michael Möller
University of Oldenburg
Michael.Moeller@informatik.uni-oldenburg.de

Abstract

Research into runtime checking of programs mainly concentrates on the Design by Contract concept, as proposed by Meyer for the programming language Eiffel. The goal is here to check whether a program fulfills certain conditions in certain states, i.e method entry and exit points. Jass (**J**ava with **ass**ertions)[1] tries to extend this to behavioural properties by adding trace assertion for dynamical checking (Jass 2). But the Jass approach is a precompiler attempt, so we cannot handle programs without its source code.

jassda, the **Jass D**ebug **A**rchitecture, is also designed to provide a trace assertion facility, but in contrast to the classic Jass 2 trace assertions these assertions are not precompiled into source code but are checked at runtime via the Java Debug Interface (JDI).

Keywords: jassda, Jass, Java, runtime checking, debugging, JDI, CSP

Introduction

Todays software systems become more and more complex. But the complexity is not the only problem to handle. Software also gets more distributed, e.g. as consequence of the growing number of web-based software systems. This results in a special interest in tracking the correctness of these software systems, but formal specification techniques and especially static checking of

[*]This work was partially funded by the German Research Council (DFG) under grant OL 98/3-1.

programs against those specification are the topic of current research (see for instance [8, 9, 14]), and currently is often not applicable for large software systems.

A practical approach to formal methods in applications is runtime checking. *Design by Contract*, as proposed by Meyer for the object-oriented language Eiffel [13], is a lightweight formal technique that allows for dynamic runtime checks of specification violations. The name refers to a contract which is made between the client and the supplier of a component and that deals with conditions before and after using such a component. The trace assertions approach extends the traditional Design by Contract concept by dealing with the dynamic order of such component uses: which component may use which other component and what third component has possibly to be used before. So we are interested in when the use of a component begins and when it ends. For this reason the entry and exit points become *events* that we want to observe, and therefore a program run emits a sequence, i.e. a *trace*, of those events.

To describe the desired behaviour of a program we define CSP-like parallel processes [7] that specify all those traces that we want to allow. In some cases certain events may not matter for the correct execution of a program, so the events of interest are restricted to the set of events mentioned in the specifying process, called the *alphabet* of the process. This alphabet helps to reduce the amount of events to be emitted by the program, since this might be an important performance aspect as [11] shows.

In contrast to the Jass 2 trace assertions [1], jassda does not precompile Java source code to emit and check events but uses the Java Debug Interface (JDI) to do this. Therefore jassda neither has to modify the source code of the observed Java program nor it has to insert event emitting Byte-code[1], in contrast to e.g. JavaMAC [10].

The following section gives an architecture overview. Section 2 will introduce the CSP-like process notion and section 3 will shortly describe the tool at work.

1. jassda architecture

For checking dynamic behaviour at runtime we need events from the program as described in the introduction. But we will only accept a minimum amount of changes to the observed Java program. Therefore our concept is to use the Java Debug Interface for emitting those events.

In the practical part of the Jass Debug Architecture we accept any information, that we can retrieve from a number of Java Virtual Machines (JVM) as events. This is almost any information that the Jass 2 trace assertion (JassTA)

[1]jassda performs slight modification on the Java Byte-code to enable full functionality

could access. Of course, the JDI provides more information than we used in JassTA because we have almost direct access to the JVM. The only informations that we cannot access directly are the return values of method calls. To overcome this limitation the Java Byte-code of the observed program is slightly modified at class loading time using the Byte Code Engineering Library [3]. Since such a JVM executes the program that we want to check (or debug), we call it *Debuggee*.

The so modified Java program is run as usual but through the JDI we set breakpoints when to emit the events that we need. So starting the program begins with connecting to the Debuggee and registering events we are interested in. Registering these events and receiving them from the JVMs is done by a central component, the jassda Broker.

In the same way as there are more than one Debuggee connected possibly to the Broker, there may also be more than one consumer for the events that the Broker provides. Figure 1 shows this architecture of jassda. These consumers of events are called jassda *modules*.

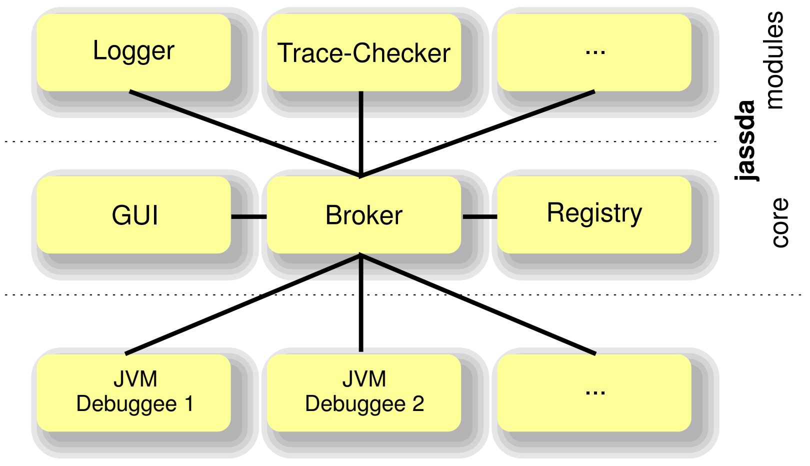


Figure 1. Architecture of jassda

One of these event listening modules is the jassda Trace–Checker. The Trace–Checker reads one or more trace specifications and builds an internal process representation for the set of legal traces (see section 2). With every received event the Trace–Checker will ensure that this actual sequence of events is a legal trace of the specification's process representation, or stop the program and inform the user of the specification violation (see section 3). Other modules may handle events in a completely different way: e.g. the Logger, an output listener, simply logs those events to a text file.

2. CSP–like processes

CSP (Communicating Sequential Processes) was introduced by Hoare [7] for the specification of event driven distributed systems. As a simple example consider the recursive equation $P = start \rightarrow stop \rightarrow P$ defining a process P that will first accept the event $start$, next it will only accept $stop$ and afterwards behave like P again. So the allowed traces of P are $\langle\rangle$, $\langle start\rangle$, $\langle start, stop\rangle$, $\langle start, stop, start\rangle$, etc. The example shows the prefix operator $\rightarrow$ prefixing a process by a single event. There are also operators for e.g. choice and parallel composition.

The Trace–Checker of jassda uses a CSP dialect, CSP_{jassda}, to specify the trace of events, that a program may emit during its execution. We chose CSP as specification language for two reasons: First, CSP is not new so that many people put research power into it, that we can now profit from. Second, our main research interest is the step from a formal specification given in CSP-OZ [4, 5], which is a combination of CSP [7] and Object-Z, to Java, and therefore it is natural to use CSP.

Since Java allows threads and (of course) multiple instances of a class, the exact specification of an event is not easy and in most cases not what we want. Therefore CSP_{jassda} uses event sets instead of single events in its prefix operator. This is the main difference between traditional CSP dialects and CSP_{jassda}, although this might be seen as syntactical shortcut for an external choice and continuing with the same process in any case.

2.1 Event Sets

Event sets in CSP_{jassda} are described by their properties. A property of an event is e.g. the thread that emitted the event, the object instance, the methods name, etc. The decision whether an event belongs to an event set is made by a handler class that must be specified in every declaration of an event set. This class could be user defined, but most cases may be covered by the class `jass.debugger.jdi.eventset.GenericSet`. The handler class will evaluate further (handler specific) properties. E.g. the `GenericSet` will check for the property `eventtype`, that could have three different values: `begin` for the entry point of a method call, `end` for normal termination and `exception` for exceptional termination of a method. Another property is `method`, that allows to specify the method's name. The configuration file allows to specify a default handler, that will be used if the handler property is missing.

In combination with the prefix operator we allow operations on event sets. This makes it convenient to describe sets of events with almost equal properties. Typically we use the intersection to express that an event must have all properties specified by the given event sets. In the syntax we use a dot or exclamation mark, because intersection is similar to CSP's communication over channels. We also allow unification of event sets indicated by a comma or plus symbol.

As another analogy to CSP's communication over channels we allow to store properties of an accepted event in a new event set, a variable, to refer to these properties in the further process. As this is like reading information from a channel, we use the question mark to indicate the binding of an event set to a variable.

2.2 Process Construction

Other basic constructs and operators of CSP_{jassda} are very similar to those of CSP_M, the CSP dialect of the model checker FDR [6]. First there are some basic processes: `STOP` indicates a deadlock[2], `TERM` indicates a terminating process[3], i.e the termination of the virtual machine, `ANY` accepts an unbounded amount of any events – often known as CHAOS.

For given Processes P, Q we currently allow prefixing ($eventset$ `->` P), choice (P `[]` Q), parallel composition (P `||` Q) and guarded (recursive) invocation of process identifiers. Parallel composition and choice are also available as "quantified versions": like binding event properties to event set variables, you may choose between processes by the property of the event.

2.3 Example

To give an impression of how program behaviour may be specified by using CSP_{jassda} processes, we will give a simple example. Consider a class `HelloWorld` with two methods `start` and `stop`. For every instance of the class we need to ensure that calls to this methods alternate, beginning with `start`.

Our first step is to declare the necessary event sets. Everything we are interested in, deals with an object of class `HelloWorld`. All events with this class property can be specified by using the predefined `GenericSet` handler.

```
eventset helloWorld
  { handler ="jass.debugger.jdi.eventset.GenericSet",
    class   ="HelloWorld" }
```

The next step is to specify the method events. Again the `GenericSet` is used to select all events that indicate the entry point (`eventtype="begin"`) of a method `start`. We do not fix the class because this is done by intersection

[2]The STOP process is introduces although no real program should ever reach this state and we can not determine if a program has reached this deadlock.
[3]most CSP dialects name it SKIP

in the process definition. Assuming that `GenericSet` is configured as default handler the event sets for `start` and `stop` are specified as follows.

```
eventset start { eventtype="begin" method="start"}
eventset stop  { eventtype="begin" method="stop"}
```

Now we want to specify, that the correct behaviour depends on the instance of the class, that produces an event. This is done by a quantified parallel operator, meaning that the following (parameterized) process is instantiated for every object instance property of an event.

```
main() {
  ||x:[instance] @ helloWorldProc(x)
}
```

The parameterized process `helloWorldProc` finally specifies the alternation of `start` and `stop`. Recall that the parameter `x` stores the instance property of the events that are directed to this subprocess.

```
helloWorldProc(x) {
  helloWorld.start.x ->
       helloWorld.stop.x -> helloWorldProc(x)
}
```

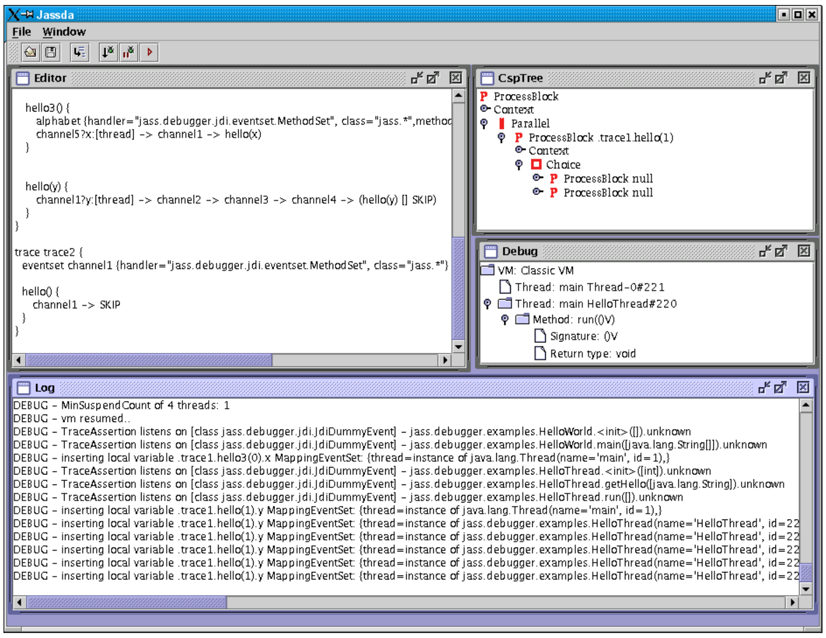
The process may be read as: "Whenever an event with `HelloWorld` as its class property arrives *and* this is a method entry event with the method property `start` *and* the instance property is the one stored in the parameter *then it must be followed by* an event with `stop` method but same instance *and then it must be followed by* the behaviour of `helloWorldProc` (i.e. the same behaviour).

3. Checking Session with jassda

A jassda session is configured by files in XML format. The main configuration file is given to jassda as command line option. This configuration file specifies the modules (with their configuration) and the connections to the debuggees.

When starting up jassda with the trace assertion module, you will get the graphical user interface, that is shown in figure 2. The main window wraps four MDI frames. One for the trace assertion specification, one for the logging output, one for the CSP representation and the last for the specification of the virtual machine.

First all but the log frames are empty. The next step is to write down the trace specification. The user edits the buffer of the specification window or loads a file into it. To check the specification, the "parse" button can be used. In case of an error, these messages go into the log frame, otherwise a graphical



```
  hello3() {
    alphabet {handler="jass.debugger.jdi.eventset.MethodSet", class="jass.*",methoc
    channel5?x:[thread] -> channel1 -> hello(x)
  }


  hello(y) {
    channel1?y:[thread] -> channel2 -> channel3 -> channel4 -> (hello(y) [] SKIP)
  }
}

trace trace2 {
 eventset channel1 {handler="jass.debugger.jdi.eventset.MethodSet", class="jass.*"}

 hello() {
   channel1 -> SKIP
 }
}
```

```
P ProcessBlock
Context
  Parallel
    P ProcessBlock .trace1.hello(1)
      Context
      Choice
        P ProcessBlock null
        P ProcessBlock null
```

```
VM: Classic VM
  Thread: main Thread-0#221
  Thread: main HelloThread#220
    Method: run(()V)
      Signature: ()V
      Return type: void
```

```
DEBUG - MinSuspendCount of 4 threads: 1
DEBUG - vm resumed..
DEBUG - TraceAssertion listens on [class jass.debugger.jdi.JdiDummyEvent] - jass.debugger.examples.HelloWorld.<init>([]).unknown
DEBUG - TraceAssertion listens on [class jass.debugger.jdi.JdiDummyEvent] - jass.debugger.examples.HelloWorld.main([java.lang.String[]]).unknown
DEBUG - inserting local variable .trace1.hello3(0).x MappingEventSet: {thread=instance of java.lang.Thread(name='main', id=1),}
DEBUG - TraceAssertion listens on [class jass.debugger.jdi.JdiDummyEvent] - jass.debugger.examples.HelloThread.<init>([int]).unknown
DEBUG - TraceAssertion listens on [class jass.debugger.jdi.JdiDummyEvent] - jass.debugger.examples.HelloThread.getHello([java.lang.String]).unknown
DEBUG - TraceAssertion listens on [class jass.debugger.jdi.JdiDummyEvent] - jass.debugger.examples.HelloThread.run([]).unknown
DEBUG - inserting local variable .trace1.hello(1).y MappingEventSet: {thread=instance of java.lang.Thread(name='main', id=1),}
DEBUG - inserting local variable .trace1.hello(1).y MappingEventSet: {thread=instance of jass.debugger.examples.HelloThread(name='HelloThread', id=22
DEBUG - inserting local variable .trace1.hello(1).y MappingEventSet: {thread=instance of jass.debugger.examples.HelloThread(name='HelloThread', id=22
DEBUG - inserting local variable .trace1.hello(1).y MappingEventSet: {thread=instance of jass.debugger.examples.HelloThread(name='HelloThread', id=22
DEBUG - inserting local variable .trace1.hello(1).y MappingEventSet: {thread=instance of jass.debugger.examples.HelloThread(name='HelloThread', id=22
DEBUG - inserting local variable .trace1.hello(1).y MappingEventSet: {thread=instance of jass.debugger.examples.HelloThread(name='HelloThread', id=22
```

Figure 2. GUI of jassda's trace assertion module

process representation will appear. Finally the "debug" button will start the Debuggee (including its JVM) and the state of its JVM will be displayed during the debugging session.

The test of the checking procedure will be that the trace of the current processes restricted to those events of the specification is included in the set of traces of the specifying process(es). Whenever an event arrives, the checker will test whether the process accepts it. If the event is not allowed, the debuggee will be stopped, otherwise the process representation will be modified in a way that the new process is a representation of the old one after accepting the event. In case of nondeterminism in the specification[4] the newly constructed process will delay choices as long as possible to keep track of all future possibilities.

4. Conclusion and Perspectives

Runtime checking of programs and especially checking Java programs is nothing new. But the concept of checking traces at runtime is a newer concept. The more frequently used technique is to emit as many events as possible, store them and do static analyses on that data. Runtime checking has the advantage

[4]This may happen in case that the event sets of a choice intersect.

that an error can be detected as soon as it happens and so it could help to reduce the harm of that error.

jassda provides runtime trace checking on byte-code level. The specifications are not related to the source code of a program, so that even third party code can be handled by the specification. The second advantage is that the user does not have to recompile her or his program for testing, or when debugging is done. This reduces the possibility of faults that are caused by preparing the code for debugging (or removing these additions).

In contrast to the trace assertions (JassTA) of Jass 2 the jassda architecture and the Trace-Checker were designed to be useful also as a stand alone tool for the "ordinary" Java programmer[5]. The JassTA have limitations, especially in distinguishing object instances, and are not thread safe. jassda overcomes these limitations and therefore might be suitable for debugging stand-alone applications, applets and servlets. Future experiences will show whether jassda is this kind of "Web-capable".

The current version of jassda was developed in a master's thesis [2] and therefore the implementation must be seen as a prototype. The most interesting question might be the question of the overhead, that is produced by the debug procedure. What is the relation between execution time with and without debugging? Another interesting question is which size of programs and which maximum number of debuggees and modules can be treated. We will try to answer these questions in the near future.

As further perspective we will work on the tool itself. We plan to add more operators and to make the specification language more comfortable. These extensions should lead us to be able to specify trace assertions that help us to check Java programs against formal specifications written in CSP-OZ [4, 5]. The goal is a mainly automated tool that will translate CSP-OZ to CSP_{jassda} and JML [12] specifications preserving the semantics.

Acknowledgments

We would like to thank the members of the Semantics Group and especially Jochen Hoenicke for a lot of interesting and productive discussions that made the current state of jassda possible.

[5]The primary design goal of JassTA was to build the counterpart of the CSP process in a CSP-OZ class.

References

[1] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – java with assertions. In Klaus Havelund and Grigore Roşu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[2] Mark Brörkens. Trace- und Zeit-Zusicherungen beim Programmieren mit Vertrag. Master's thesis, University of Oldenburg, January 2002. in German.

[3] Markus Dahm. Byte Code Engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, April 2001.

[4] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, IFIP, pages 423–438. Chapman & Hall, 1997.

[5] C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR 2*, Dec. 1995. Manuscript.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *FASE 2000: Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284 – 303. springer, 2000.

[9] K. Huzing, R. Kuuiper, and SOOP. Verification of object-oriented programs using class invariants. In T. Maibaum, editor, *FASE 2000: Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 208 – 221. springer, 2000.

[10] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[11] David Kortenkamp, Tod Milam, Reid Simmons, and Joaquin Lopez Fernandez. Collecting and analyzing data from distributed control programs. In Klaus Havelund and Grigore

Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[12] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical report, Department of Computer Science, Iowa State University, 1998, revised 2001.

[13] B. Meyer. *Object-Oriented Software Construction*. ISE, 2nd edition, 1997.

[14] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000. (to appear).