Electronic Notes in Theoretical Computer Science 70 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 15 pages

Dynamic Event Generation for Runtime Checking using the JDI [1]

Mark Brörkens [2]

OFFIS, R & D Division Embedded Systems
Escherweg 2, D–26121 Oldenburg, Germany

Michael Möller [3]

University of Oldenburg, Department of Computer Science
Postbox 2503, D–26111 Oldenburg, Germany

Abstract

Approaches to runtime checking have to track the execution of a software system and therefore have to deal with generating and processing execution events. Often these techniques are applied at the code level – either by inserting new source code prior to the compilation or by modifying the target code, e.g. Java byte code, before running the program.

The jassda [4,3] framework and tool enable runtime checking of Java programs against a CSP-like specification. For generating events it uses the Java Debug Interface (JDI) and thus no modifications to the code are necessary. Another advantage is that events are generated on demand, i.e. dynamically at runtime it is determined which events to generate for the current debug run without modifying the program itself. This paper shows how this event generation is done by the jassda framework.

1 Introduction

Software systems have grown substantially since the invention of micro computers. This trend cannot only be seen in the size of programs, but also in application areas. The growth of the Internet with more and more dynamic web-pages providing new services for end-users is another category of this complexity. Today's software systems are distributed, work in parallel and have to communicate.

[1] This work was partially funded by the German Research Council (DFG) under grant OL 98/3-1.
[2] Email: Mark.Broerkens@offis.de
[3] Email: Michael.Moeller@informatik.uni-oldenburg.de

In all these software systems correctness becomes more important but is harder to prove due to the growing complexity. Traditional methods to prove this correctness are *Modelchecking* [5] and *Program Verification* [8,1]. The development of program verification techniques for object-oriented languages (and its support by theorem provers) is a topic of current research (see for instance [9,10,18]) and have been successfully applied in domain specific areas of software development. However, these techniques are often only viable with specific knowledge of underlying theorem provers and are therefore restricted to experts.

Runtime checking is a *lightweight formal technique* that will undoubtedly not exceed the strength of the above *heavyweight techniques*. In this case only the current run of the program is checked against the specification, but it is easier to apply to (even large) systems and to be used by non-experts.

For the Java programming language many runtime checking approaches concentrate on the *Design by Contract* concept as proposed by Bertrand Meyer for the language Eiffel [16]. The name refers to a contract which is made between the client and the supplier of a component. The contract states the obligations of the client before using a method and the constraints provided by the supplier after use of the service. The sytax of these conditions, called *assertions*, is usually close to the programming language. Tools that provide this for Java are e.g. jContractor [11], iContract [14] or the runtime-checker of JML [15]. The Jass[4] tool [2] in addition provides *Trace Assertions* that allow to state conditions describing the order of *events*, i.e. method entry and exit points.

In the *Trace-Checker* of jassda[5] these Trace Assertions are extended to support Java programs in a more general way. The event emitting mechanism of Jass was not satisfying, since the event emitting code for every possibly interesting event had to be inserted. As consequence a lot more events than needed were generated and events for classes without source code could not be generated. Dynamic "on the fly" generation of events is important to reduce the huge amount of possible events like in Morphine [6]. But to modify the *system-under-test* (SUT) as less as possible, jassda uses the *Java Debug Interface* (JDI) to let the virtual machine generate these events. As a positive side-effect it is also possible to monitor distributed systems of communicating Java programs, because a number of Java virtual machines may be connected.

The next section will describe the architecture of the jassda framework to explain the use of the JDI. The following sections describe the JDI and the event model used in jassda. Further a benchmark, a conlusion and an overview of related work is given.

[4] **J**ava with **ass**ertions
[5] **Jass** **D**ebugger **A**rchitecture

2 The jassda Framework – Motivation

The jassda tool allows the runtime checking of a system of Java programs against a CSP-like specification. In this specification language events have to be consumed and it has to be decided whether they are conforming with the specification. To get those events from the system-under-test (SUT), i.e. a system of Java programs, a general event extraction and dispatching facility was developed, the jassda framework. This framework might also be used for other purposes, e.g. just logging events [6], or to stimulate a program for test purposes.

The architecture of this framework is shown in figure 1. At the lowest level the *debuggees* are shown, which together form the system-under-test. These debuggees are connected to the Broker which is the central component of the jassda framework. The "Registry" database, an optional graphical user interface and the Broker build the jassda core. jassda modules are connected to this core requesting and consuming events.

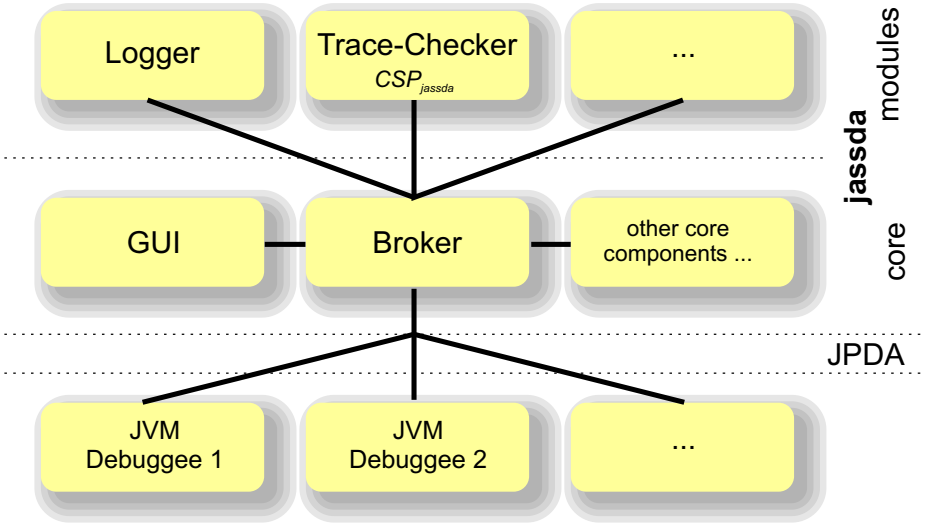

Fig. 1. Architecture of jassda

The connection between the debuggees and the jassda core transports the events that we want to observe [7]. This connection is established by using the Java Platform Debugger Architecture (JPDA) as described in the next section.

3 Generating Execution Events

One of the aims of the design, when developing the jassda tool, was to achieve a method for monitoring Java programs reduced to a minimum of modifications

[6] This application of the framework is implemented by the "Logger-Module"
[7] Additionally it will also send back control information

to the program itself. To achieve this, the Java Debug Interface (JDI) [19] is used.

3.1 The Java Debug Interface (JDI)

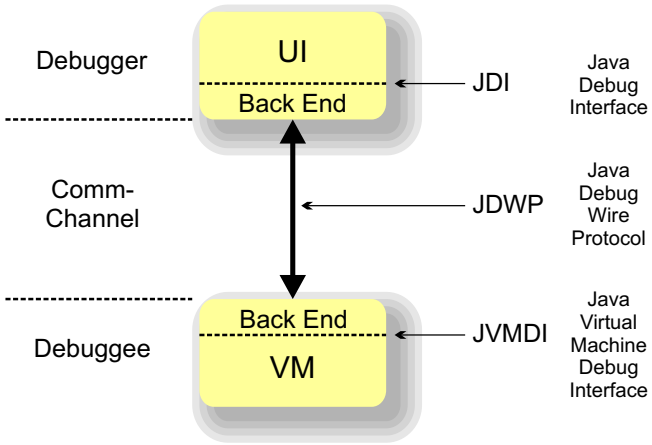

Fig. 2. Java Platform Debugger Architecture

Current Integrated Development Environments for Java like Sun Forte for Java, Borland JBuilder or IntelliJ IDEA contain a debugger, based on the Java Platform Debugger Architecture (JPDA) for communicating with the system that is to be debugged. The JPDA consists of two programming interfaces, i.e. the Java Virtual Machine Debug Interface (JVMDI) and the Java Debug Interface (JDI), and one protocol, i.e. the Java Debug Wire Protocol (JDWP), as shown in figure 2. The JVMDI is a low-level native interface that defines the services which a Java virtual machine must provide for debugging. The second interface, the JDI, provides a high-level Java programming language interface. The format of information and requests transferred between the debuggees and the debugger Front-End is defined by the JDWP.

In addition to well known functions like setting breakpoints and watching variables, the JDI provides an extended facility for monitoring and manipulating the execution of a Java program.

During runtime the debuggees can be configured to generate events in case of several situations, e.g. a method has started or terminated, an exception has occurred, a breakpoint is reached, a class is loaded/unloaded, read/write access to a variable, a thread was started/stopped. After having emitted an event the debugging VM can be configured to suspend execution and thus allow a deep view into the VM. For example: for each currently running thread its stack trace can be analysed. For each class the inner structure like super-classes and implemented interfaces can be read. Even the byte code of every method can be accessed for further analysis.

3.2 jassda and the JDI

As shown above, the JDI allows generating events for many situations that might occur during the execution of a Java system. The main function of the core layer of the jassda framework is to determine which events are required for the analysis of the given Java system and to configure the debuggees for the generation of those events, only. The set of events required for a debug run depends on the installed modules and their configurations. The execution of a debug run with the jassda framework can be separated into two phases.

3.2.1 Initial Phase

In the initial phase the jassda framework connects to the debuggees using the JDI and suspends their execution. Then the loaded classes are statically analysed in order to determine the set of events that might be generated during their execution. After that, the modules are asked which of these events they are interested in. Accordingly, the generation of those events is configured using the JDI.

3.2.2 Runtime Phase

After having configured the events, the execution of the debuggees are resumed. The execution is suspended again as soon as a situation of interest occurs in one of the debuggees and an event is generated. The broker receives this event and distributes it to the modules for further processing. Through the data transferred with the event the modules have full access to the JDI and therefore may perform any action the JDI supports, e.g. they can read and manipulate the values of variables or find out which situation triggered the current event.

3.2.3 Handling Dynamic Class Loading

As Java supports loading classes dynamically during runtime, events that notify about loading classes are enabled. In this way the jassda framework keeps informed about new classes and it configures the event generation for them, as described for the initial phase.

3.2.4 Obtaining Return-Values of Methods

For many utilisations of runtime-checking obtaining the return-value of a method is of great importance. Unfortunately, the JDI does not support access to return-values of methods, directly. For the jassda framework we developed a solution to find out that value without having to modify the JDI and therefore loose the benefit of the platform-independence.

The main idea for accessing the return-value is to write the value into a private variable and generate an event immediately before the method terminates. To achieve this, the jassda framework provides a special class-loader that patches the byte code of any method for which a module requires the

return-value during runtime.

4 Processing Execution Events

The jassda framework provides an infrastructure for developing debugger-applications that have to monitor the execution of Java systems. A high-level interface enables the developer to plug in one or more modules for processing the events delivered by the framework. These modules must implement two basic functions. On the one hand they have to decide which events they are interested in, i.e. the *alphabet*. This is required by the core layer to configure the generation of events from the debuggees, accordingly. On the other hand they need to handle the sequence of events delivered by the broker one by one during runtime.

4.1 Modules and Alphabets

To determine the alphabet of a registered module, the module has to provide request methods that are used by the Broker. During startup and whenever a new class is loaded the Broker asks each module if it is interested in events emitted during execution of that class. In case of positive result the Broker requests a list of event types [8] that the module wants to receive. Concerning this list the Broker analyses the new class and creates dummy events for all events that could belong to the module's alphabet. In the final step the module decides for all these possible events which do belong to its alphabet. This decision is used by the Broker to configure the debuggees for event emitting and for dispatching the events delivered by the debuggees during runtime.

The architecture of the jassda framework includes the capability of reconfiguration of the event emitting mechanism. New events can be triggered. Enabled events can be disabled during the debug run. This feature will be used in future versions of the jassda tool.

Currently there are two modules available: the *Logger module* and the *Trace-Checker module*. Together with the jassda framework they form the jassda tool.

4.2 Logger Module

The Logger module implements the function to log the execution of a Java system by writing its sequence of events into a file. The amount of information that can be derived from an event as well as the alphabet can be configured in an XML-based configuration file.

Figure 3 shows such a configuration. The output is directed to the file `sequence.log` in the format given by the `template` definition. In this definition field-identifiers are replaced by the property values of the received events.

[8] These event types are taken from the JDI, e.g. method entry, method exit, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<logger>
  <file name="sequence.log"/>
  <event>
    <template>%class%.%method%(%arguments%) = %returnvalue%</template>
  </event>
  <include>
    <eventset class="jass.debugger.jdi.eventset.GenericSet"
              field="class"
              argument="jass.examples.*"/>
  </include>
  <exclude>
    <eventset class="jass.debugger.jdi.eventset.GenericSet"
              field="class"
              argument="sun.*"/>
  </exclude>
</logger>
```

Fig. 3. Configuration of the Logger module

The events to include and to exclude are determined by handler classes defined through the `class` property of `eventset`. The `GenericSet` handler, that comes with jassda will be sufficient in most cases, but one may also write their own event filters.

It would be easy to write similar modules that write those events to a database via JDBC, the Java database connection. In any case the logged data can be used for analysing the program, e.g. in the way it is done by Kortenkamp et al. in [13]. Therefore when using this module, jassda performs comparable operations like other event collecting tools.

4.3 Trace-Checker Module

More comfortable is the use of a CSP_{jassda} specification to not only receive events but in addition to analyse the received events on the fly. This trace checking is very similar to trace assertions in Jass, but it is generalised to a more expressive language. A more detailed view on the CSP dialect CSP_{jassda} is given in [17] and in the appendix, but with a small example we will illustrate how to determine the Trace-Checker's alphabet from a given specification.

In figure 4 the normal behaviour of applets is specified. The first lines define event sets. Event sets are used to group events. Often we do not know all properties of an event or some aspects are not relevant in a specification. But the alphabet of the module is not directly built by the union of these event sets.

The alphabet of the specification is built by the alphabet of the first process in the specification – in this example the alphabet of `applets()` has to be used. To calculate this alphabet all events mentioned in this process are collected. Since this process is built on a parallel composition of the process

```
eventset applet  { instanceof="java.applet.Applet" }
eventset init    { method="init" }
eventset start   { method="start" }
eventset stop    { method="stop" }
eventset destroy { method="destroy" }

applets() {
  ||i:[instance] @ appletbehaviour(i)
}

appletbehavior(inst) {
  applet.inst.init.begin -> applet.inst.init.end
   -> appletrun(inst)
}

appletrun(i) {
  ( applet.i.start.begin -> applet.i.start.end
     -> applet.i.stop.begin  -> applet.i.stop.end -> aplletrun(i)
  )  [] appletdestroy(i)
}

appletdestroy(inst) {
  applet.inst.destroy.begin -> applet.inst.destroy.end -> STOP
}
```

Fig. 4. CSP_{jassda} specification "applet behaviour"

`appletbehaviour`, it is necessary to take the union of all alphabets of the subprocesses for each object instance.

The definition of `appletbehaviour` starts with the first event set that we have to add to the alphabet. `applet.inst.init.begin` defines an event set that is built by the intersection of the four parts. `applet` defines the set of all events that are emitted by classes that are an instance of `java.applet.Applet`. `init` defines all events fired by methods named `init`. `begin` is a predefined event set describing all events of the method begin type. The event set `inst` is the parameter of the process, it is undefined when calculating the alphabet and will not restrict the event set. So this first expression defines the set of events that are fired whenever a method `init` of any instance of an applet begins.

Continuing with this calculation we will end up in an alphabet of all events triggered by invocation and normal termination of the methods `init`, `start`, `stop` and `destroy` for all instances of classes extending `Applet`. To forbid certain events during the debug run it is also possible to state the alphabet of a process explicitly.

5 Benchmark Example

In order to gain information about the performance of the jassda tool checking Java applications, a small program is used which sorts a list of 10000 numbers using the bubblesort algorithm. This program provides two implementations of the algorithm. The first one calculates in one method and therefore requires one method invocation. The second implementation puts the inner loop into a separate method and requires 10001 method invocations. The bubblesort program is executed by the Sun Java 1.3 virtual machine (in classic and hotspot mode) and by the Sun 1.4 virtual machine under Windows 2000 on a Pentium (1200 MHz) system. Three different configurations are compared:

(i) The program is executed *standalone* without any monitoring.

(ii) The Java virtual machine is configured to run in *debug-mode*

(iii) The Java virtual machine is configured to run in debug-mode and the *jassda tool is attached* logging method invocations into a file.

method invocations (events)		1	10001
standalone	j2sdk1.3 classic	9,2s	9,2s
	j2sdk1.3 hotspot	1,0s	1,2s
	j2sdk1.4 hotspot	1,0s	1,1s
debug-mode	j2sdk1.3 classic	39,0s	39,0s
	j2sdk1.3 hotspot	10,7s	10,7s
	j2sdk1.4 hotspot	1,1s	1,1s
jassda attached	j2sdk1.3 classic	40,2s	65,3s
	j2sdk1.3 hotspot	11,2s*	1805,3s*
	j2sdk1.4 hotspot	11,2s*	1804,0s*

* with less than 1% CPU usage

Table 1
Benchmark results

As shown in table 1 the sorting of 10000 numbers in the *standalone* configuration requires 9,2s in classic mode and round about 1s in hotspot mode. Enabling the *debug-mode* increases the execution time of the Java 1.3 VM by up to 10 times. The execution time of the Java 1.4 VM keeps almost constant, due to its full-speed debugging support. *Attaching the jassda tool* reduces the performance of the Java 1.3 VM in classic mode compared to the debug-mode by factor 1.5 . The last two lines show that the execution time dramatically increases if the virtual machine is in hotspot mode and has to handle breakpoints which the jassda tool uses for indication of method invocations. In this case the CPU usage is below 1% whereas otherwise 100% CPU usage is required. The reason for the low CPU usage and long execution time is still

beeing examined.

6 Conclusion

In this paper we presented the mechanism of event emitting used in the jassda framework. The implementation of a prototype has proven that this is a viable method to debug some small Java programs. But these experiments have also shown the limitations of our approach.

The jassda framework works fine, if we can reduce event generation through the dynamic concept of "on demand" or "on the fly" generation. To debug an efficient algorithm with very limited code to execute between needed events will reduce the performance dramatically.

But the aim was to have a method to debug distributed systems so that the focus lies on communication events. In contrast to Jass the motivation for the jassda tool was not to be able to perform Design by Contract runtime checks – although this might possibly be done by new modules in future versions. Whereas Design by Contract assertion build the counterpart of a state based specification the jassda framework should enable us to build a counterpart for a dynamic process based specification. For this reason we see jassda more as an addition to state based approaches than as a replacement.

In addition the limitations can also be seen as an advantage: a run with low amount of events is cheap and in the same way it is cheap to switch the set of events for a second run. Only the initial phase is influenced by the new set of events, but no byte-code modification is necessary even though only the needed events for the second run will be generated. So we presume that the jassda framework will have a considerable advantage in environments where event sets are limited but do often change.

6.1 Related Work

The idea to reduce the amount of events to those that are needed in the current debug run was already introduced in Morphine [6] for programs written in PROLOG. But the code for the event filtering for Java programs is provided by the Java virtual machine (VM) of the system-under-test, so that jassda does not have to modify the byte code to insert statements or function calls for event generation.

For the same reason jassda differs from Java-MaC [12], that performs exactly that byte code modification. A *primitive event definition language* is used to define events of interest. Using this definition event generating code is inserted where needed. For this reason a new event specification means a second patching of the Java byte code.

The idea of an event definition language, that separates the definition of events from the source code, was also given by Gates et al. in [7]. But again this technique uses automated program instrumentation to generate events

and thus does not generate them on demand.

6.2 Future Work

Beside a translation from a combined specification language to the input language of the jassda Trace-Checker, we plan to clean up the source code and provide it as an Open Source project via the jassda homepage [9] .

References

[1] Apt, K.-R. and E.-R. Olderog, "Verification of Sequential and Concurrent Programs," Springer-Verlag, 1997, 2nd edition.

[2] Bartetzko, D., C. Fischer, M. Möller and H. Wehrheim, *Jass – Java with Assertions*, in: K. Havelund and G. Roşu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, Electronic Notes in Theoretical Computer Science **55** (2001).

[3] Brörkens, M., "Trace- und Zeit-Zusicherungen beim Programmieren mit Vertrag," Master's thesis, Universität Oldenburg (2002), in German.

[4] Brörkens, M. and M. Möller, *jassda Trace Assertions*, in: I. Schieferdecker, H. König and A. Wolisz, editors, *Trends in Testing Communicating Systems*, International Confernece on Testing Communicating Systems (TestCom), Berlin, Germany, 2002, pp. 39–48.

[5] Clarke, E., E. Emerson and A. Sistla, *Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach*, in: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1983, pp. 117–126.

[6] Ducasse, M. and E. Jahier, *Efficient automated trace analysis: Examples with morphine*, in: K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, Electronic Notes in Theoretical Computer Science **55** (2001).

[7] Gates, A. Q., S. Roach, O. Mondragon and N. Delgado, *Dynamics: Comprehensive support for run-time monitoring*, in: K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, Electronic Notes in Theoretical Computer Science **55** (2001).

[8] Hoare, C. A. R., *An axiomatic basis for computer programming*, Comm. of the ACM **12** (1969), pp. 576–580.

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

[9] `http://jassda.sourceforge.net`

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

[11] Karaorman, M., U. Hölzle and J. Bruno, *jContractor: A Reflective Java Library to Support Design By Contract*, Technical report, Department of Computer Science, University of California, Santa Barbara (1998), `http://www.cs.ucsb.edu/~murat/jContractor.PDF`.

[12] Kim, M., S. Kannan, I. Lee, O. Sokolsky and M. Viswanathan, *Java-mac: a run-time assurance tool for java programs*, in: K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, Electronic Notes in Theoretical Computer Science **55** (2001).

[13] Kortenkamp, D., T. Milam, R. Simmons and J. L. Fernandez, *Collecting and analyzing data from distributed control programs*, in: K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, Electronic Notes in Theoretical Computer Science **55** (2001).

[14] Kramer, R., *iContract - the Java Design by Contract tool*, Technical report, Reliable Systems (1998), `http://www.reliable-systems.com`.

[15] Leavens, G., 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).

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

[17] Möller, M., *Specifying and Checking Java using CSP*, in: *Workshop on Formal Techniques for Java-like Programs - FTfJP'2002*, Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 2002, (to appear).

[18] Müller, P. 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).

[19] Sun Microsystems, *Java Platform Debugger Architecture Documentation* (1999), `http://java.sun.com/products/jpda/doc/`.

A CSP_{jassda} Semantics

In this appendix we will briefly describe the basic elements of CSP_{jassda} and define the operational semantics, that is used during runtime checking.

A.1 Event Sets

Event sets are defined by a number of properties, i.e. key value pairs, enclosed in curly braces. A property named `handler` is required to specify a Java

handler class that is responsible for deciding whether a JDI event belongs to the current set or not. Other properties a read by the handler class to configure it.

For most cases the class `GenericSet`, that comes with the jassda Trace-Checker will do fine. Therefore this is the default if no handler was specified. The current implementation of `GenericSet` will accept a number of properties, e.g. to specify the class, the method, etc. User defined classes are also allowed to be used as handler classes and therefor will have full access to the JDI events.

By using the `eventset` keyword event sets are bound to the identifier that follows the keyword.

We allow intersection ("`.`" or "`!`") and union ("`+`") of event sets. This allows one to state that the current event should have the properties of both event sets or it should have the properties of at least one event set.

Finally new event sets can be defined "on the fly" during the debug run. Whenever an event is accepted (see next sections), and thus is a member of a current event set, the properties of the current event may be used to define the new event set. This is done by adding a "`?`" to the event set definition followed by the variable identifier and the mapping. Again, this mapping is done by a handler class, with `MappingEventSet` as default implementation.

To give an example we assume that `first` and `second` are defined event sets. Than an event set `first.second?third:[arg0="arg1"]` will accept all events that belong to both event sets, `first` and `second`, and when an event is accepted a new event set will be defined with identifier `third`, that will match all events with a first argument that equals the second argument of the current event.

A.2 Processes

Currently the jassda Trace-Checker accepts two basic processes: `STOP` and `TERM` where `STOP` will accept no event and `TERM` will only accept termination events of the debuggees' virtual machines.

By *prefixing* a process with an event set we define a new process, that first will only accept an event belonging to the event set and then behave like the prefixed process.
Syntax:

$$Process \ ::= \ eventset \texttt{ -> } Process$$

Semantics:

$$\frac{}{es \texttt{ -> } P \xrightarrow{ev} P} \quad \text{where } ev \in es$$

External choice will split split up the behaviour into two branches but avoids nondeterminism.
Syntax:

$$Process \ ::= \ Process_1 \texttt{ [] } Process_2$$

Semantics:

$$\frac{P \xrightarrow{ev} P', \neg\exists Q' : Q \xrightarrow{ev} Q'}{P \,[]\, Q \xrightarrow{ev} P'} \qquad \frac{Q \xrightarrow{ev} Q', \neg\exists P' : P \xrightarrow{ev} P'}{P \,[]\, Q \xrightarrow{ev} Q'}$$

$$\frac{P \xrightarrow{ev} P', Q \xrightarrow{ev} Q'}{P \,[]\, Q \xrightarrow{ev} P' \,[]\, Q'}$$

Quantified external choice will bind a new event set variable.
Syntax:

$$Process \;::=\; []\, var : [map]\; Process_1(var)$$

Semantics:

$$\frac{P(x) \xrightarrow{ev} P'(x)}{[]\, v : [map]\, P(v) \xrightarrow{ev} P'(x)} \quad \text{where } x = map(ev)$$

Parallel composition is always synchronised over the intersection of alphabets of both processes. We will use $\alpha(P)$ as notion for the alphabet of process P (the events a process is interested in).
Syntax:

$$Process \;::=\; Process_1 \,||\, Process_2$$

Semantics:

$$\frac{P \xrightarrow{ev} P'}{P \,||\, Q \xrightarrow{ev} P' \,||\, Q} \quad \text{where } ev \in \alpha(P) \setminus \alpha(Q)$$

$$\frac{Q \xrightarrow{ev} Q'}{P \,||\, Q \xrightarrow{ev} P \,||\, Q'} \quad \text{where } ev \in \alpha(Q) \setminus \alpha(P)$$

$$\frac{P \xrightarrow{ev} P', Q \xrightarrow{ev} Q'}{P \,||\, Q \xrightarrow{ev} P \,||\, Q'} \quad \text{where } ev \in \alpha(Q) \cap \alpha(P)$$

Analogous to external choice we define a *quantified parallel composition*. Semantically it is a parallel composition where for each new mapping result we will get a new process instance.
Syntax:

$$Process \;::=\; ||\, var : [map]\; Process_1(var)$$

Semantics:

$$||\, v : [map]\, P(v) \quad = \quad ||^{\emptyset}_{v:map} P(v)$$

$$\frac{P(x) \xrightarrow{ev} P'(x)}{||^{D}_{v:map} P(v) \xrightarrow{ev} P'(x) \,||\, ||^{D \cup x}_{v:map} P(v)} \quad \text{where } x = map(ev) \wedge x \cap D = \emptyset$$

Whenever a named (parameterised) processes is defined the process identifier may be used where a process is expected.
Syntax:

$$ProcessDefinition \;::=\; Id(Params, \ldots)\{[Alphabet;]Process\}$$

$$Proccess ::= ProcessId(Params, \ldots)$$

Semantics:

$$\frac{P \xrightarrow{ev} P'}{Id(p) \xrightarrow{ev} P'} \quad \text{where } process(Id(p)) = P$$

A.3 Trace Semantics

The jassda Trace-Checker will test during runtime if the *trace* of the program's events belongs to the trace semantics of the specification. The process of the first process definition in the specification defines this semantics.

The trace semantics of a process P is defined by collecting all event sequences that are possible with respect to the operational semantics. So the empty sequence $\langle\rangle$ is always included and every initial event extended by any trace of the subsequent process.

$$traces(P) = \{\langle\rangle\} \cup \{\langle ev\rangle^\frown tr | \exists P' : P \xrightarrow{ev} P' \wedge tr \in traces(P')\}$$