Appendix B

Trace File Format

An execution trace is stored in a ZIP file. This ZIP file contains a single file, called `trace`, which stores the actual trace data in a textual, line-oriented format. This format is described further on.

By using a ZIP file and a textual, line-oriented format, a trace can be opened and read by a programmer without using specialized tools. A ZIP utility and a text editor suffice. This has simplified the development of the trace generator and parser. Furthermore, a ZIP file is easily extensible, because additional information about the trace can be stored by adding files to the ZIP file.

A disadvantage of this approach is that, without compression, the file `trace` is very large. This situation is improved by using the compression offered by a ZIP file. Another disadvantage is that, due to the combination of ZIP decompression and the parsing of a large file, loading a trace takes a long time.

Nowadays, it is a common choice to base a textual format on XML. XML is a good format to store complex hierarchical data. It is well known in the industry and many tools exist to deal with it. XML formats add a lot of structural information (tags). This structural information is not strictly necessary for the `trace` file, because the structure of the data it stores (a list of events) is fairly simple. It would only increase the size the `trace` file, without significantly improving readability. Furthermore, the format described in the next section is simpler than an XML format would be. For this project, it was considered best to use the simplest format that could store the needed data. Hence, the `trace` file uses a custom format, which is not based on XML.

B.1 The `trace` File

The file `trace` is a text file (i.e. it is human readable), encoded as UTF-8. It stores a list of events, separated by newlines. All events have the following header:

```
<event type>:<time stamp>
```

where

- `<event type>` is the event type, encoded as two capital letters, and
- `<time stamp>` is the event's time stamp, the value of the system timer at the time of the event.

Information specific to an event type can be included after the header. The event types are described in the following sections.

VM Start Event

A VM Start Event indicates the start of the Java Virtual Machine. It is always the first event in a trace. Its syntax is:

```
VS:<time stamp>
```

VM Init Event

A VM Init Event indicates that the Java Virtual Machine is initialized and ready to execute the program. It is always the second event in a trace. Its syntax is:

```
VI:<time stamp>
```

VM Death Event

A VM Death Event indicates that the Java Virtual Machine has terminated. It is always the last event in a trace. Its syntax is:

```
VD:<time stamp>
```

Class Load Event

A Class Load Event indicates that the Java Virtual Machine has loaded a class. Its syntax is:

```
CL:<time stamp>:<class name>
```

where `<class name>` is the fully qualified name of the loaded class. The class name is stored as it is in a class file, i.e., the identifiers that make up the fully qualified class name are separated by an ASCII forward slash ('/') (See [17, Section 4.2]). For instance, the class `java.lang.Object` (as used in the Java programming language) is stored as `java/lang/Object`.

Object Allocation Event

An Object Allocation Event indicates that an object has been created. Its syntax is:

```
OA:<time stamp>:<class name>:<object id>
```

where

- `<class name>` is the name of the created object's class, and
- `<object id>` is a unique number identifying the object.

Object Free Event

An Object Free Event indicates that an object has been freed by the garbage collector. Its syntax is:

```
OF:<time stamp>:<class name>:<object id>
```

where

- `<class name>` is the name of the created object's class, and
- `<object id>` is a unique number identifying the object.

Thread Start Event

A Thread Start Event indicates that a thread has been started. Its syntax is:

```
TB:<time stamp>:<thread id>
```

where `<thread id>` is a unique number identifying the thread.

Thread Stop Event

A Thread Stop Event indicates that a thread has been stopped. Its syntax is:

```
TE:<time stamp>:<thread id>
```

where `<thread id>` is a unique number identifying the thread.

Method Entry Event

A Method Entry Event indicates that a thread has entered a method. Its syntax is:

```
MN:<time stamp>:<thread id>:<class name>:<method name>:<object id>
```

where

- `<thread id>` is the identifier of the thread entering this method,
- `<class name>` is the name of the class defining the entered method,
- `<method name>` is the name of the entered method, and
- `<object id>` is the identifier of the object referenced to by `this` in the current frame. It is 0 if the entered method is static.

Method Exit Event

A Method Exit Event indicates that a thread has left a method. Its syntax is:

```
MX:<time stamp>:<thread id>:<class name>:<method name>
```

where

- `<thread id>` is the identifier of the thread leaving this method,
- `<class name>` is the name of the class defining the left method,
- `<method name>` is the name of the left method, and

Frame Pop Event

A Frame Pop Event indicates that a frame has been popped due to an exception. Its syntax is:

```
FP:<time stamp>:<thread id>:<class name>:<method name>
```

where

- `<thread id>` is the identifier of the thread in which the frame was popped,
- `<class name>` is the name of the class defining the popped method, and
- `<method name>` is the name of the popped method.