Formats of FITTEST Log Files

A. Elyasov A. Middelkoop I.S.W.B. Prasetya
J. Hage
Dept. of Inf. and Comp. Sciences, Utrecht Univ.
{A.Elyasov,A.Middelkoop,S.W.B.Prasetya,J.Hage}@uu.nl

August 8, 2011

1 Raw Log File

A raw log file is produced directly by the target program. It has an XML-like structure. In general a log is a sequence of log-entries. Each entry is organized as a 'section', which in itself can be recursively made of sub-sections. The lowest level section is called 'paragraph', which consists of a list of sentences. A sentence is basically just a string. There is no format impossed, however a specific logger may impose a certain syntax on the sentences. Sections are tagged, which can be used to associate some semantic to them.

The grammar is below.

```
Character set: UTF-8
Extension: .log

<Log> ::=  (<Section> <white*>)+

The first section is a header section. It is used to describe the
application, its user, date, etc.

<Section>      ::= <SectionStart> <white*> (<SectionPart> <white*>)* <End>
<SectionStart> ::= %<S <white*> <TimeStamp>? <white*> "<tag>"  -- note that tag has to be quoted!
<End>          ::= %>

<SectionPart>  ::= <Para> | <Section>
<Para>         ::= %<P <white*>  (<Sentence> <white*>)* <End>
<Sentence>     ::= %<{ <SentenceContent> }%>
<SentenceContent> ::= <any>*  -- must not contain the combination }%>

Time stamp is formatted as follow:

     <UTC-offset> ":" <current-UTC-time>

The current time is expressed in terms of UTC time (so, not depending on time zone)
and is encoded as a single integer, which is the difference between the current
UTC-time and UTC midnight 1-st Jan. 1970 in milli-seconds.

The time in terms of local time can be inferred from the given UTC-offset. The offset
has this format:

   <UTC offset>   ::= ("+" | "-")? <offset-in-minutes>
```

2 Specialized Raw Format

The basic raw-log format just defines the concept of sentence, paragraph, section. For most logging purpose we will want to have a richer semantic of the logs. This can be done by further specifying the raw-log syntax. For example by defining a syntax for the sentences and the section tags. Below are a number of specialized formats that we use.

2.1 Serialization Format

This defines how objects are printed out in the log (how they are 'serialized' to the log). Our concept of 'object' broadly represents a structure of some data. It can be a real object in the target program, or it can be a primitive value. Or, it can represent a fake object which the target program uses to wrap over some information to pass to a logger.

We will make a distinction between 'simple' and nested objects. A simple object is an object that will be formatted as a single-sentence paragraph. A primitive value will be formatted like this. Other objects are considered 'nested' and will be formatted as a section, with the needed nesting.

2.1.1 Single Paragraph Object

```
The general format is: %<P %<{ <value>:<type> }%> %>
?? is used to denote unserializable value.

Undefined and null:

  %<P %<{ undefined:void }%> %>
  %<P %<{ null:Null }%> %>

Int and uint:

  %<P %<{ 199:int }%> %>

Number:

  %<P %<{ 0.00000123:Number }%> %>

Bool and string:

   %<P %<{ false:Boolean }%> %>
   %<P %<{ "hello world!":String }%> %>

Object that the serializer does not know how to handle:

   %<P %<{ ??:eu.fittest.Logging.Serialization::TEST_DefaultSerializer2_simple_cases }%> %>
```

2.1.2 Nested Object

```
The general format is:

   %<S "O:<object-type>"  <object-part>* %>

   <object-part> ::= %<P <field>+ %>
                  | %<P <field>* <ref-field> %> <nested-object>

   <field> ::= %<{ <field-name>=<value>:<type> }%>
            | %<{ <field-name>=^<id-number> }%>   -- reference to an object
```

```
The syntax of the <value>:<type> part above is the same as in the
single paragraph object (see above).

A ref-field is a field whose value is another nested object. This object
is serialized just after the field (see above syntax).

   <ref-field> ::=  %<{ <field-name>=> }%>

The first field of a nested object is always a bogus field added by the serializer
to contain an id. This id is a unique number identifying the object within the
serialization (so it is not unique over multiple calls to serialization).
It role is to handle the serialization of an object with a cycle inside
its structure. The format of this ID-field:

  %<{ I:<id-number>:ID }%>

Some examples:

An object with int and Number fields:

    <S "O:ExamplePair"
    %<P
    %<{ X:0 }%>
    %<{ fst=199:int }%>
    %<{ snd=0.00000123:Number }%>
    %>
    %>

A object with cyclic structure :

    %<S "O:ExamplePair"
    %<P
    %<{ I=0:ID }%>
    %<{ fst=> }%>
    %>
    %<S "O:ExamplePair"
    %<P
    %<{ I=1:ID }%>
    %<{ fst=null:Null }%>
    %<{ snd=^0 }%>
    %>
    %>
    %<P
    %<{ snd=null:Null }%>
    %>
    %>
```

2.2 Log functions format

The Logging package offers a set of functions to produce logs. Log fragmets are supplied to these functions in terms of the usual programming elements. In particular, it hides the details about the used log format, so that programmers do not have to worry about it. They generate raw-log format, and refine the raw syntax as follows.

```
Logging application state:

   %<S <time-stamp>? "S" <object>  %>
```

Logging flash-event:

```
  %<S <time-stamp>? "E" <event-object> <app-state-object>  %>
```

Logging function entry

```
  %<S <time-stamp>? "FE:<func-name>:<class-name>"
       <target-obj>
       %<S "args"  <object>* %>
  %>
```

If the function does not have a target-obj, then it is just null.

Logging function exit:

```
  %<S <time-stamp>? "FX:<func-name>:<class-name>"
      <target-obj>
      <return-obj>
  %>
```

Logging function-call entry:

```
   %<S <time-stamp>? "FCE:<caller-func-name>:<caller-class-name>"
       %<P %<{ <callee-func-name>:<callee-class-name> }%> %>
       <target-obj>
       %<S "args"  <object>* %>
   %>
```

Logging function call exit:

```
   %<S <time-stamp>? "FCX:<caller-func-name>:<caller-class-name>"
       %<P %<{ <callee-func-name>:<callee-class-name> }%> %>
       <target-obj>
       <return-obj>
       <exception-obj>
   %>
```

Logging block:

```
   %<S <time-stamp>? "B:<block-id>:<func-name>:<class-name>" %>
```

Logging exception handler:

```
   %<S <time-stamp>? "BEH:<block-id>:<func-name>:<class-name>"
      <exception-obj>
   %>
```

Logging loop enter:

```
   %<S <time-stamp>? "BLE:<block-id>:<func-name>:<class-name>"%>
```

Logging loop exit:

```
   %<S <time-stamp>? "BLX:<block-id>:<func-name>:<class-name>"
       %<P %<{ cnt=1000001 }%> %>
   %>
```

3 Indexed Log

We do not actually store logs in their raw format. Instead, we convert them to an indexed format. This format is more compact in a number of ways:

1. timestamps are represented more compactly.
2. section tags are indexed. This means that duplicated tags will be replaced by a single integer representening the index of the tag. Of course this also means that we also have to keep a dictionary of indices.
3. paragraphs are also indexed.

This has the benefit that if we want to filter a log based on the timestamps and the tags of its entries, we do not need to expand the whole original raw log.

4 XML Log File

For processing by other tools we can convert our logs to XML format. The format will have this general form:

```
<?xml version='1.0' ?>
<body>
... entry-1
... entry-2
...
</body>
```

4.1 Application Event

An 'application event' is a log entry that represents a single user interaction with the target program. E.g. if it was the user clicking on a button, the entry will say so and also specifies which button is clicked. If we provide a way to extract the state of the program, this state will be sampled when the event happens, and logged. Since the full state of a program is usually very large, one should provide a way to abstract it. This should somehow be hooked to the used logger.

The structure of 'event' is as follows:

```
<E timestamp>
    ... event description
    ... target program's state
</E>
```

The event description describes the event (of course), and it has this structure:

```
<O ty="eu.fittest.actionscript.automation::RecordEvent">
  <fd n="I"> ... just an object counter </fd>
  <fd n="targetID"> ... specifies which GUI element is interacted on  </fd>
  <fd n="type">     ... specifies what action done on that GUI </fd>
  <fd n="args">     ... the action may require arguments       </fd>
</O>
```

So, 'action' can be 'click' or 'type'. If it was click then 'args' will contain no argument. If it was 'type' then 'args' will contain the text that was typed. Because 'args' may hold multiple arguments, it will be represented as an array:

```
<fd n="args">
  <O ty="Array">
     <fd n="I"> ... another object counter  </fd>
     <fd n="elem"> <V v="999" ty="int" /></fd>
     <fd n="elem"> ... perhaps more element</fd>
     ...
 </O>
</fd>
```

Here is a full example of 'event' entry:

```
<E t="-120:1312787896474">
    <O ty="eu.fittest.actionscript.automation::RecordEvent">
      <fd n="I">
        <V v="0" ty="ID" />
      </fd>
      <fd n="targetID">
        <V v="&quot;ButtonBar0&quot;" ty="String" />
      </fd>
      <fd n="type">
        <V v="&quot;itemclick&quot;" ty="String" />
      </fd>
      <fd n="args">
        <O ty="Array">
          <fd n="I">
            <V v="1" ty="ID" />
          </fd>
          <fd n="elem">
            <V v="1" ty="int" />
          </fd>
        </O>
      </fd>
    </O>
    <O ty="AppAbstractState">
      <fd n="I">
        <V v="0" ty="ID" />
      </fd>
      <fd n="numOfSelectedItems">
        <V v="18" ty="int" />
      </fd>
      <fd n="numInShopCart">
        <V v="0" ty="int" />
      </fd>
      <fd n="cartCurrency">
        <V v="&quot;$&quot;" ty="String" />
      </fd>
      <fd n="cartTotal">
        <V v="&quot;$0.00&quot;" ty="String" />
      </fd>
    </O>
</E>
```

4.2 Deep-Logging Entries