FITTEST Log Format (version 1.1)

I.S.W.B. Prasetya

A. Elyasov

A. Middelkoop

J. Hage

Technical Report UU-CS-2012-014
Sept. 2012

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

FITTEST Log Format (version 1.1)

Wishnu Prasetya Alexander Elyasov Arie Middelkoop Jurriaan Hage

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

1 Short Introduction

This report describes the FITTEST log format. "FITTEST" is the name of the project from where the format originates, which stands for Future Internet Testing. The format itself has nothing to do with Internet; it can be used to express any kind of logs.

The important feature of the FITTEST format is that is deeply structured ala XML. However, it is more compact (in our examples, by a factor of 2). As in XML, log entries are tagged. Tags give additional meaning to them. We will impose on a set of base tags[1].

There are two ways FITTEST logs can be produced:

1. By writing a logger that natively produces FITTEST logs.
2. By translating logs in an existing format to the FITTEST format.

There is an accompanying command line program called `haslog` that can convert FITTEST logs to an XML format so that people can use XML tools to inspect the logs, e.g. we can then write XPath/XQuery queries on the logs.

Haslog can also "compress" logs. It may or may not result in smaller logs. The compression exploits values in the FITTEST logs that are grouped in paragraphs. The effectiveness of this depends on how smart the used logger did the grouping.

Note: additionally, the FITTEST project has some tools to analyze FITTEST logs (or their XML versions), though these tools are still experimental. E.g. there is a tool to infer finite state automata from logs. Haslog also has an experimental feature to convert FITTEST logs to the Daikon format. This allows test oracles in the form of pre- and post-conditions to be inferred from the logs by using the Daikon tool [1]. □.

Overview

Section 2 describes the syntax of FITTEST logs. Section 3 describes the base tags. It also specifies how events and objects should be serialized in the logs. Section 4 describes the structure of the corresponding XML versions of FITTEST logs.

2 Syntax

FITTEST logs are in **UTF-8** rather than binary. They are stored in files with **.log** extensions.

A FITTEST log is just a sequence of *log entries*. Each entry is organized as a *section*, which can recursively be made of sub-sections. The lowest level section is called *paragraph*, which consists of *sentences*. A sentence is basically just a string. There is no format imposed (however a specific logger

[1] Historically, the format was first used to log ActionScript programs. This has some influence in some choices in the base tags.

may impose a certain syntax on the sentences —we will return to this later). Sections are tagged, which can be used to associate some semantic to them. The syntax is below.

⟨*log*⟩ ::= (⟨*section*⟩ ⟨*whites*⟩)+

⟨*section*⟩ ::= ⟨*section start*⟩ ⟨*whites*⟩ (⟨*section part*⟩ ⟨*whites*⟩)* ⟨*end marker*⟩

⟨*section start*⟩ ::= %<S ⟨*whites*⟩ ⟨*time stamp*⟩? ⟨*whites*⟩ "⟨*tag*⟩"
— note that a tag has to be quoted.

⟨*end marker*⟩ ::= %>

⟨*whites*⟩ ::= ⟨*white*⟩*

⟨*section part*⟩ ::= ⟨*paragraph*⟩ | ⟨*section*⟩

⟨*paragraph*⟩ ::= %<P ⟨*whites*⟩ (⟨*sentence*⟩ ⟨*whites*⟩)* ⟨*end marker*⟩

⟨*sentence*⟩ ::= %<{ ⟨*sentence content*⟩ }%>

⟨*time stamp*⟩ ::= ⟨*UTC offset*⟩ : ⟨*UTC time*⟩

⟨*UTC offset*⟩ ::= (+ | -)? ⟨*offset in minutes*⟩

Additional constraints:

1. **Sentence**

   ⟨*sentence content*⟩ is any sequence of character, but it should not contain the combination }%>, which is used to identify the sentence's end.

2. **Time stamp**

   Time stamp is written as a pair $o : t$ where t is UTC time, which is location independent, and o is the offset of the local time with respect to the UTC time. With this offset we can infer what the local time of t is.

   The UTC-time t encoded as a single integer, which expresses the number of milli-seconds elapsing since midnight 1-st January 1970 and the actual UTC time when t is measured.

   The offset o is also an integer, expressing the time difference between the location on which t is measured and the UTC time, expressed in minutes.

   This choice was influenced by ActionScript.

A note about compression

Organizing logs in paragraphs and sentences gives opportunity for compression. E.g. we can collect sentences that occur repeatedly in a log, and then index them in a dictionary. Then we can replace the occurrences of a sentence s with its index, and thus making the log more compact, while to some extent still searchable. We can do the same with long section-tags or even paragraphs that occur repeatedly. Timestamps can be compressed as well. Only sections are timestamped. If S is a section with timestamp t, we can instead represent it as the difference with the previous timestamp, which usually takes less space to store.

The tool `haslog` can do such compression. The command:

```
haslog -c hello.log
```

will produce `hello.lox` and `hello.dic`; the latter is the dictionary. We then need to apply a generic compressor (e.g. zip) on these results.

Compression is however not part of the FITTEST format.

3 Base Tags

This section define a set of base tags used by the FITTEST format. A subset of these tags are used to describe events. FITTEST format also allow objects' state to be logged, so the other subset of tags are related to object serialization.

We will use the following meta notation for describing the tags' syntax:

1. Sen(x) means that x will be formatted as a sentence, with x as the content.
2. Par($s_1 ... s_n$) means that this will be formatted as a paragraph with each s_i forming a sentence of the paragraph, appearing in the order as specified.
3. Sec($T, S_1 ... S_n$) means that this will be formatted as a section with tag T, with the S_i's concatenated to form the body of i-th section. This section has **no** timestamp.
4. Sec$_{timed}$($T, S_1 ... S_n$) is as above, except that the section can be timestamped.

3.1 Object Format

Here we define how objects should be printed/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 a fake object that we use to abstractly represent a complicated object.

Because a FITTEST log is defined as a sequence of events, objects are never logged on their own. They must be part of events. The tagging of events is describe in Subsection 3.2.

We will have two kinds of objects: simple and nested. A *simple object* or 'value' only has a single paragraph with a single sentence.

⟨*object*⟩ ::= ⟨*simple object*⟩ | ⟨*nested object*⟩

⟨*simple object*⟩ ::= Par(Sen(⟨*simple value*⟩))

⟨*simple value*⟩ ::= ⟨*undefined*⟩ | ⟨*null*⟩ | ⟨*integer*⟩ | ⟨*numeric*⟩ | ⟨*boolean*⟩ | ⟨*string*⟩ | ⟨*unserializable*⟩

⟨*undefined*⟩ ::= `undefined:void`
⟨*null*⟩ ::= `null:Null`
⟨*integer*⟩ ::= ⟨*int value*⟩`:int`
⟨*numeric*⟩ ::= ⟨*numeric value*⟩`:Number`
⟨*boolean*⟩ ::= ⟨*boolean value*⟩`:Boolean`
⟨*string*⟩ ::= ⟨*string value*⟩`:String`
⟨*unserializable*⟩ ::= `??:`⟨*class name*⟩

Additional notes and constraints:

1. There is no special format for natural numbers (non-negative integers).
2. A numeric value is either an integer or a dot-separated float value like 1.05.
3. A boolean value is either `true` or `false`.
4. A string value has to be quoted, e.g. `"hello world"`.
5. A class name can be simple of fully qualified[2].
6. `Number` and `undefined` were influenced by ActionScript.

[2] ActionScript a fully qualified name of a class `Item` looks like this `eu.fittest.mypackage::Item`

```
%<P %<{ undefined:void }%> %>
%<P %<{ null:Null }%> %>
%<P %<{ 199:int }%> %>
%<P %<{ 0.00000123:Number }%> %>
%<P %<{ false:Boolean }%> %>
%<P %<{ "hello world!":String }%> %>
%<P %<{ ??:eu.fittest.MyPackage::Item }%> %>
```

Figure 1: Examples of simple objects.

3.1.1 Nested Object

A nested object has multiple fields, some of these may contain subobjects. These fields are grouped in one or more paragraphs, which can be exploited for compression (see the note in Section 2). E.g. the logger may group the fields such that those that tend to vary together are put in the same paragraph. In any case, the logger is free to decide how to arrange the fields in paragraphs; but they must be packed in paragraphs.

There is however one restriction imposed by the syntax. Consider as an example this class:

```
class Person {
   var name : String ;
   var age  : int ;
   var spouse : Person ;
   var height : int ;
}
```

The spouse field of a person contains another person as a subobject. This cannot be packed in a paragraph, and has to be packed as a section instead. We will however put the field name (spouse) in the paragraph that precedes it. For example, here is how a person can be serialized:

```
%<S "O:Person"
  %<P  %<{ I=0:ID }%>
       %<{ name="Sponge Bob":String }%>
       %<{ age=4:int }%>
       %<{ spouse=> }%>  // the spouse-object can be found in the next section
  %>
  %<S "O:Person"
     ... // here is the serialization of the spouse
  %>
%>
```

Figure 2: An example of a nested object. The field `I` above is "a fake field"; this will be explained later. The indentations are not part of the syntax. The syntax allows them (white space), and we add them in all examples for readability. In a real log we probably do not want to add them to save space.

Below is the syntax of nested objects:

⟨*nested object*⟩ ::= Sec(⟨*object tag*⟩, ⟨*object body*⟩)
⟨*object tag*⟩ ::= 0:⟨*class name*⟩
— Note that an object tag is a section tag, so it has to be quoted.
⟨*object body*⟩ ::= ⟨*object part*⟩*

⟨*object part*⟩ ::= Par(⟨*simple field*⟩$_1$ ... ⟨*simple field*⟩$_n$)
— at least one field, but see also the constraint below.
| Par(⟨*simple field*⟩$_1$ ... ⟨*simple field*⟩$_n$ ⟨*subobject marker*⟩) ⟨*nested object*⟩
— n can be 0, but see also the constraint below.
⟨*simple field*⟩ ::= Sen(⟨*field name*⟩ = ⟨*simple value*⟩)
| Sen(⟨*field name*⟩ =^ ⟨*iref*⟩)
⟨*subobject marker*⟩ ::= ⟨*field name*⟩ =>

Constraint: the fake field `I` must be the first field of a nested object. The next section explains the purpose of `I`.

3.1.2 Object with Cyclic Structure

Suppose we have an object o to serialize into the log. When the subobjects structure beneath o contains a cycle, the logger cannot just recursively serialize o; this will result in an infinitely large log. So, as it traverses o, it need to keep track of the object references it saw, and number them. Let's call these numbers *local indices*. As it logs o and its subobjects, for each it also need to log the corresponding local index in the fake field `I` like what we have seen in the example Figure 2. The value of this field is an integer, but is marked as having the type `ID`.

When the logger comes to a field in o that points to an object p it has seen before, it should not serialize the object. Instead it should write p's local index. For example, consider an object p of the class Person such that its spouse points to x itself. This could be serialized as follows:

```
%<S "O:Person"
  %<P  %<{ I=0:ID }%>
       %<{ name="Patrick":String }%>
       %<{ age=4:int }%>
       %<{ spouse=^0 }%>  // refers to local index 0
  %>
%>
```

Figure 3: An example of a cyclic object.

The `I` indices are called 'local' because they are only need to be unique within the traversal of a given top-level object that we are serializing (in other words, they are not unique over the whole log). So, if after Patrick we write another person Bob to the log, Bob's local indices may again start from 0.

3.1.3 Array, Collection, and Dictionary

An array or collection a is treated as an ordinary object of type `Array` respectively `Collection`. Each element x of a is treated as a field called `elem` (so, we will have N fields called `elem`). If a is an array, the elements will be serialized the same order as in the array.

A dictionary d is treated as an object of the type `Dictionary`. Each entry $(key, value)$ in the dictionary will be treated as two consecutive fields called `key` and respectively `val`.

3.2 Event Format

We distinguish between high level and low level events. High level events are typically used to represent user interactions with the target application (or in general, interactions of other types of external

```
%<S "O:Array"
  %<P  %<{ I=0:ID }%>
       %<{ elem=10:int }%>
       %<{ elem=100:int }%>
  %>
%>
```

```
%<S "O:Dictionary"
  %<P  %<{ I=0:ID }%>
       %<{ key=0:int }%>
       %<{ val=10:int }%>
       %<{ key=1:int }%>
       %<{ val=100:int }%>
  %>
%>
```

Figure 4: The first is an example of an array, it has two elements: 10 and 100. The second is a dictionary containing two entries $(0, 10)$ and $(1, 100)$

agents), such as when a user click on a GUI button. Low level events are typically used to represent events occurring within the application itself, such as when some internal method/function f is being called.

3.2.1 High level events

A high level event e is described by an 'event object' and a state object. The first describes what kind of event e is (e.g. a click on a button, or an update to a text-field), and the second abstractly describes the state of the target application 'when' the event occurs. We do not impose here what the exact temporal relation between the sampled state and the event; this depends on the implementation of the logger. Typically, the state is sampled after the event occurs, and before the next event occurs.

⟨*high level event*⟩ ::= Sec_{timed}(E, ⟨*event object*⟩ ⟨*state object*⟩)
⟨*event object*⟩ ::= ⟨*nested object*⟩
⟨*state object*⟩ :: = ⟨*object*⟩

The event object must have the following fields, see the example in Figure 5:

1. `type:String`, specifies what kind of event that is. E.g. `"itemclick"` means the event was a click on some display object.

2. `targetID:String`, is a string that identifies the display object that is targeted by the event. E.g. if the event was a click on a button b, this this would be the ID-name of b. This ID is usually unique.

   Note: one can potentially analyze the high level events in logs to construct a model describing valid sequences of high level events. In the FITTEST project, such a model is used to generate test-cases. Note that for such a purpose the ID of event targets *must* be unique, or else the test driver cannot know where to target a replayed event to.

3. `args:Array`, is an array containing the arguments passed to the event. Event like clicking on a button has no argument. On the other hand, an update to a text-field is an event that takes one argument, namely the new value in the text-field.

```
%<S -120:1312787896474 "E"
  %<S "O:eu.fittest.actionscript.automation::RecordEvent"
      %<P %<{ I=0:ID }%>
          %<{ targetID="ButtonBar0":String }%>
          %<{ type="itemclick":String }%>
          %<{ args=> }%>
      %>
      %<S "O:Array"
          %<P  %<{ I=1:ID }%>
               %<{ elem=1:int }%>
          %>
      %>
  %>
  %<S "O:AppAbstractState"
     %<P  %<{ I=0:ID }%>
          %<{ numOfSelectedItems=18:int }%>
          %<{ numInShopCart=0:int }%>
          %<{ cartCurrency="$":String }%>
          %<{ cartTotal="$0.00":String }%>
     %>
  %>
%>
```

Figure 5: *An example of a high-level event representing a user click on* `ButtonBar0`*. The event object is of the class* `RecordEvent`

3.2.2 Low level events

The FITTEST provides events that can be used to log the entrance and exit of functions, and visits to instructions blocks inside functions. Examples are shown in Figure 6.

The events of type `FE` (*function entry*) and `FX` (*function exit*) are used to log at the entrance and exits of a function. The syntax is below:

⟨*function entry*⟩ ::= Sec_{timed}(⟨*function entry tag*⟩, ⟨*target object*⟩ ⟨*args*⟩)
⟨*function entry tag*⟩ ::= `FE:`⟨*function name*⟩

⟨*function exit*⟩ ::= Sec_{timed}(⟨*function exit tag*⟩, ⟨*target object*⟩ ⟨*return object*⟩)
⟨*function exit tag*⟩ ::= `FX:`⟨*function name*⟩

⟨*function name*⟩ ::= ⟨*function name*⟩`:`⟨*class name*⟩
⟨*return object*⟩ ::= ⟨*object*⟩

Suppose this is used to log a function f, which can be called as $o.f(x)$. We call o the *target object*. So the above syntax allows this object to be serialized into the log.

The syntax of the target object and arguments are:

⟨*target object*⟩ ::= ⟨*object*⟩
⟨*args*⟩ ::= Sec(`args` ⟨*object*⟩*)

The program of a method can be divided into 'blocks'. A block is a maximal consecutive segment of instructions that does not contain a jump nor targeted by a jump. The events below can be dispatched to log when a block is visited.

⟨*visit block event*⟩ ::= Sec_{timed}(⟨*visit block tag*⟩,)
⟨*visit block tag*⟩ ::= `B:`⟨*block id*⟩`:`⟨*function name*⟩

```
%<S -120:1347473178132 "FE:move:Point"
    %<S "O:Point"
        %<P %<{ I=0:ID }%>
            %<{ x=10:int }%>
            %<{ y=10:int }%>  %>
    %>
    %<S "args"
        %<P %<{ 2:int }%> %>
        %<P %<{ 3:int }%> %>
    %>
%>
```

```
%<S -120:1347473178141 "B:564:move:Point" %>
%<S -120:1347473178141 "B:632:move:Point" %>
%<S -120:1347473178142 "B:633:move:Point" %>
%<S -120:1347473178143 "B:642:move:Point" %>
```

```
%<S -120:1347473178143 "FX:move:Point"
    %<S "O:Point"
        %<P %<{ I=0:ID }%>
            %<{ x=12:int }%>
            %<{ y=13:int }%>  %>
    %>
    %<P %<{ undefined:void }%> %>
%>
```

Figure 6: *An example of a pair of* FE *and* FX *events describing* move(2,3) *where* move *is a function/method of the class* Point*. In the example, the arguments are simple values, but in principle they can also be objects. In the middle we see block events that corresponds to the instructions-blocks the execution of the call* move(2,3) *internally pass. The block events are not mandatory, they are showed here just as example.*

3.2.3 Planned low level events

The following low level events were part of the original definition of the FITTEST format. However, they are currently not supported by our FITTEST tools.

Rather than logging the entrance and exit of a function f inside f itself, we can alternatively log this at the calls to f. This allows us to also log who the caller is and the exception thrown by f. On the other hand, this requirescall call locations to f to be discovered (so that we can instrument them for logging). This can be non-trivial, e.g. due to the combination of dynamic binding and information erasure at the bytecode level.

Let f be the calling function (the caller), and suppose it calls $g(x)$ (callee). The events FCE and FCX can be used to log when the call begins (entry) and ends (exit). The format of the latter allows exception thrown by the callee to be logged. If the used logger actually does this, then it will have to be able to catch and rethrow g's exceptions. These details are of course the logger's decision; here we simply describes the syntax of how those events shoul be described in the log.

⟨*function call entry*⟩ ::= Sec_{timed}(⟨*function call entry tag*⟩, ⟨*callee*⟩ ⟨*target object*⟩ ⟨*args*⟩)
⟨*function call entry tag*⟩ ::= FCE:⟨*function name*⟩
⟨*callee*⟩ ::= Par(⟨*function name*⟩)
⟨*function name*⟩ ::= ⟨*function name*⟩:⟨*class name*⟩
⟨*target object*⟩ ::= ⟨*object*⟩
⟨*args*⟩ ::= Sec(args ⟨*object*⟩*)

⟨*function call exit*⟩ ::= Sec_{timed}(⟨*function call entry tag*⟩, ⟨*callee*⟩ ⟨*target object*⟩ ⟨*return object*⟩ ⟨*exception object*⟩)
⟨*function call entry tag*⟩ ::= FCX:⟨*function name*⟩
⟨*exception object*⟩ ::= ⟨*object*⟩

The events below can be dispatched to log when an exception handler is entered, when a loop is entered, and when a loop is exited.

⟨*visit block tag*⟩ ::= `B:`⟨*block id*⟩`:`⟨*function name*⟩

⟨*handling exception event*⟩ ::= Sec_{timed}(⟨*handling exception tag*⟩, ⟨*exception object*⟩)
⟨*handling exception tag*⟩ ::= `BEH:`⟨*block id*⟩`:`⟨*function name*⟩

⟨*enter loop event*⟩ ::= Sec_{timed}(⟨*enter loop tag*⟩,)
⟨*enter loop tag*⟩ ::= `BLE:`⟨*block id*⟩`:`⟨*function name*⟩

⟨*exit loop event*⟩ ::= Sec_{timed}(⟨*exit loop tag*⟩, Par(⟨*iteration count*⟩))
⟨*exit loop tag*⟩ ::= `BLX:`⟨*block id*⟩`:`⟨*function name*⟩
⟨*iteration count*⟩ ::= `cnt=`⟨*integer*⟩

4 XML Log File

The FITTEST format is XML like, but it is more compact. This is nice for loggers, because they need to minimize I/O. On the other hand, for analysis we may still prefer XML because of the availability of various XML-based tools.

Using the commands below we can convert a FITTEST log to the XML format. It will produce the file `hello.xml`.

```
haslog -c hello.log
haslog -x hello.log
```

This section describes the structure of the resulting XML file. The elements this XML is will be described notations like this:

element E =
attrib optional t
content ⟨*event object*⟩ ⟨*state object*⟩

This defines an element with E as the tag, and it has an optional attribute called t, and has two sub-elements: an event object and a state obecjt. An incomplete example of an instance of this element is here:

```
<E t="-120:1312787896474">
   ...  // event object
   ...  // state object
</E>
```

The top-level element is body, which consists of entries as sub-elements. An entry can be a high level event, or a low level event.

element body = **content** ⟨*entry*⟩*

⟨*entry*⟩ = ⟨*high level event*⟩ | ⟨*low level event*⟩

⟨*high level event*⟩ = E

Here is the structure of a high level event:

element E = — high level event
attrib optional t — time stamp
content ⟨*event object*⟩ ⟨*state object*⟩

⟨*event object*⟩ = O
⟨*state object*⟩ = ⟨*object*⟩

```
<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="&amp;quot;ButtonBar0&amp;quot;" ty="String" />
      </fd>
      <fd n="type">
        <V v="&amp;quot;itemclick&amp;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="&amp;quot;$&amp;quot;" ty="String" />
      </fd>
      <fd n="cartTotal">
        <V v="&amp;quot;$0.00&amp;quot;" ty="String" />
      </fd>
    </O>
</E>
```

Figure 7: *A log in the XML format.*

Conform to the constraint in Subsection 3.2.1, an event object should consist of these fields: `type:String`, `targetID:String`, and `args:Array`. Figure 7 shows an example (in XML).
An object can be nested or simple (value):

⟨*object*⟩ = O | V

element O = — nested object
  attrib type — the object's type
  content fd* — fields

element fd = — object's field
  attrib n — field name
  content (O | V) — field's content

element V = — simple object
  attrib v — the object's value
  attrib ty — the object's type

The syntax for low level events is shown below.

⟨*low level event*⟩ = FCE | FCX | FE | FX | B | BEH | BLE | BLX

element FCE = — function call entry event
  attrib f — caller function name
  attrib ce — callee name
  content ⟨*target object*⟩ args

⟨*target object*⟩ = ⟨*object*⟩

element args = **content** ⟨*object*⟩*

element FCX = — function call exit event
  attrib f — caller function name
  attrib ce — callee name
  content ⟨*target object*⟩ ⟨*return object*⟩ ⟨*exception object*⟩

⟨*return object*⟩ = ⟨*object*⟩
⟨*exception object*⟩ = ⟨*object*⟩

element FE = — function entry event
  attrib f — function name
  content ⟨*target object*⟩ args

element FX = — function exit event
  attrib f — function name
  content ⟨*target object*⟩ ⟨*return object*⟩

element B = — visit block event
  attrib f — function name
  attrib i — block ID

element BEH = — handling exception event
  attrib f — function name
  attrib i — block ID
  content ⟨*exception object*⟩

element BLE = — enter loop event
  attrib f — function name
  attrib i — block ID

element BLX = — exit loop event
  attrib f — function name
  attrib i — block ID
  attrib cnt — iteration count

References

[1] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.