Action Script Instrumentation

October 24, 2010

1 Introduction

Our goal is to instrument the execution of an Action Script program in order to obtain an abstract representation of transitions of the program's state during its execution. Concretely, we are interested in an execution trace that contains some information about the methods executed and their respective parameters. The programs under considerations are Adobe Flash programs written using Action Script version 3.

An Action Script program is compiled to byte code, which in turn is interpreted by a virtual machine to execute the program. This provides us with three possibilities to instrument an Action Script program:

- Instrument the virtual machine (Section 2). This provides us with limitless access to the run-time state of the machine (foreign code aside), and does not require us to preprocess the program. On the other hand, since a Action Script program typically runs in a web browser, it may not be possible to change the virtual machine running in that browser.
- Instrument the byte code (Section 3). With this approach, we can instrument any program or library that we explicitly pass to the virtual machine. The standard library, which is already present in the virtual machine, cannot be instrumented. This, however, is not a real limitation in practice. Advantageous is that the instrumentation is part of the program, which ensures that it automatically works fine in combination with optimizations performed by the virtual machine, such as just-in-time compilation.
- Instrument the source code. The sole advantage is that information that we have access to information not preserved by the compilation process, such as the precise lexical structure of nested expressions and loops. Some information, such as the names of local variable and some source code locations are optionally preserved in the byte code, when compiling to byte code with debugging enabled. This approach is impractical: we cannot instrument libraries or programs of which we do not have the source code.

The Action Script language evolved from Javascript to Java. Like Java, a program is modeled as classed with methods, with code-reuse via inheritance. The classes define statically what the structure of an object is, and what methods an object has. An object is associated statically with a class (the compile-time type), and dynamically with a subclass of that class (the run-time type). Classes that are explicitly marked as dynamic may have additional structure at runtime not described by the class. This effectively corresponds to a hidden association list stored with each object, containing additional fields of the class. Such an object also has a field named `prototype`, which points to an object whose fields and methods the current object shared (and possibly overrides).

Another dynamic feature of Action Script is the possibility to use a run-time computed string as name of a variable. Combined with the possibility to query the query the structure of an object, this provides a reflection API to the programmer. These dynamic features, however, complicate the instrumentation or make it less efficient.

2 Virtual Machine Instrumentation

The virtual machine of Action Script (version 3), called AVM2, is standardized[1]. The Tamarin-project[2] of Mozilla provides an open-source virtual machine written in C++ that interprets the byte code.

The sources of the virtual machine can be obtained via the Mercurial[3] version control software from:

```
hg clone http://hg.mozilla.org/tamarin-redux/
```

There are several ways to build the source code. On Mac OSX Snow Leopard, it is possible to use Eclipse. First update to a recent version of Eclipse with the CDT extension installed. The build-settings are for a previous version of the operating system. Update paths that point to the OSX 10.4 SDK to the 10.6 SDK. Also, add `/usr/lib` to the library path, and `z` (zlib) to the library list.

Todo. Info on structure of the interpreter. In what file to find the main loop. Info on the main data structures of the interpret.

3 Byte-code Instrumentation

Todo. The ASC compiler.

[1]http://www.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf
[2]http://www.mozilla.org/projects/tamarin/
[3]http://mercurial.selenic.com/

4 Instrumentation language

4.1 Layer 1: General Instrumentation

Todo.

```
e  ::=  x           -- reference the value bound to x (may be a qualified nar
   |    e @ e       -- application
   |    \x . e      -- abstraction
   |    d ; e       -- bind
   |    e >> e      -- match the current program point against the lhs
   |    e # e       -- alternatives (order left to right)
   |    retain      -- given an x and e, memorize x in the context e
   |    action      -- if driven by a match, given an x, executes that exter

   |    any         -- arbitrary value
   |    class       -- a class, given e
   |    field       -- a field, given e
   |    method      -- a method, given e
   |    local       -- a local, given e
   |    param       --

   |    within      -- within a certain context e
   |    call        -- call to a certain context e
   |    assign      -- assignment, given x and e
   |    deref       -- dereference, given e
   |    type        -- a type, given x and value e

d  ::= x = e            -- definition (non-recursive)
   |   store x = e      -- named store
   |   data x = (x+)+   -- data definition
```

4.2 Layer 2: Tracing