Fine-grained patches for Java software upgrades

Eduardo R. B. Marques
Faculdade de Ciências da Universidade de Lisboa
edrdo@di.fc.ul.pt

Abstract

We present a novel methodology for deriving fine-grained patches of Java software. We consider an abstract-syntax tree (AST) representation of Java classes compiled to the Java Virtual Machine (JVM) format, and a difference analysis over the AST representation to derive patches. The AST representation defines an appropriate abstraction level for analyzing differences, yielding compact patches that correlate modularly to actual source code changes. The approach contrasts to other common, coarse-grained approaches, like plain binary differences, which may easily lead to disproportionately large patches. We present the main traits of the methodology, a prototype tool called aspa that implements it, and a case-study analysis on the use of aspa to derive patches for the Java 2 SE API. The case-study results illustrates that aspa patches have a significantly smaller size than patches derived by binary differencing tools.

1 Introduction

Echoing Lehman's law of continuous change [10], modern software is evolving constantly and software upgrades are routinely deployed. Consider for instance "smartphone apps", where upgrades are very common and directly affect end user experience in many ways, like data transfer and associated cost, installation time, or user intervention. Thus, software upgrades must be increasingly reliable, efficient, and automated, both for the end user and the software vendor or provider.

Upgrades are defined by software *patches*, reflecting the transition between software versions. A patch p between old and new versions O and N of a software artifact in compiled form is such that $N = p(O)$. That is, the patch p must encode the transformation from O to N, that occurs as part of the upgrade from O to N in a target platform. It is many times the case that p is not an incremental transformation of O, but instead amounts to the entire N, i.e., $p = N$, corresponding to the full installation of the new version, e.g., as in Android smartphone apps. More refined, incremental approaches define p as the set of changed files from O to N, or attend to the binary differences between O and N or between component files within O and N [1, 16, 17, 19].

All of the above approaches are common. The problem is that they are too coarse-grained and operate at an inappropriate abstraction level. In the general case, a resulting patch p may not appropriately reflect the may have a disproportionate size to the "actual" changes between O and N, i.e., the relevant syntactical differences between O_S and N_S.

We propose a better solution to this problem, in the context of Java software [7] compiled to the Java Virtual Machine [12] bytecode format. The approach is to account for the *fine-grained changes* between two versions O and N of a JVM class file, expressed in an abstract syntax tree (AST) representation. The JVM class file format is closely related to the core traits of Java in source form. Hence JVM-level patches may potentially correlate more evenly with source-code level changes, whilst avoiding the obvious inconveniences of using source-code based patches for upgrades (e.g., a Java compiler on the target platform, IP issues). On the other hand, since the JVM format is for all purposes still a binary one, an AST representation may factor out features that are syntactically irrelevant or do not correlate to source code changes, e.g., the definition order of methods in a JVM class file or constant pool indexes spread throughout it [12].

We have developed a prototype tool called aspa that implements this methodology, written in Java and available from [13]. In the remainder of the paper, we begin by the describing the main traits of the methodology and the aspa tool (Section 2). We then present results of using aspa over the core Java 2 SE (J2SE) API, showing that aspa patches can be much smaller than binary difference patches (Section 3). We end the paper with a

discussion of related work, highlights for future work, and possible use of the presented methodology in other contexts (Section 4).

2 The aspa tool

Overview. Our approach is illustrated in Fig. 1. Given old and new versions of a Java class in the binary JVM format, C_O and C_N, the aspa tool parses both to derive corresponding symbols $A_O = \mathsf{ast}(C_O)$ and $A_N = \mathsf{ast}(C_N)$ with an AST-like representation. A patch for the upgrade from C_O to C_N is derived by computing AST-level differences between A_O and A_N, $p = \mathsf{diff}\ (A_O,\ A_N)$. The patch p can then be applied to A_O to obtain A_N, i.e., $A_N = p(A_O)$, after which A_N can be converted back to the JVM format, $C_N^p = \mathsf{jvm}(A_N)$.

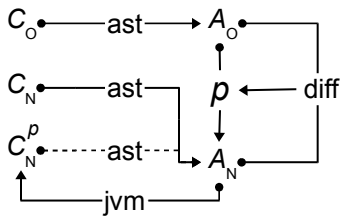

Figure 1: The aspa approach

Note that in Fig. 1 we can have that $C_N^p \neq C_N$ in terms of the binary JVM format, but in any case $\mathsf{ast}(C_N) = \mathsf{ast}(C_N^p) = A_N$ at the AST level. This is a by-product of the core trait of aspa: A_O, A_N and p factor out a number of serialization aspects in the JVM binary format that are syntactically irrelevant, but lead to disproportionate binary-level differences between C_O and C_N, e.g., the indexes of constants in the JVM pool or method definition order [12]. Binary differences are by definition sensitive to these aspects, but aspa factors them by out by appropriate mechanisms in AST data representation and difference analysis, described next.

AST representation of JVM class files. In Fig. 2 we depict a fragment of the abstract syntax grammar embedded into aspa to represent a Java class, using semi-formal BNF notation. The grammar abstracts the core symbolic information found in a JVM class file [12] and, in close relation, also the fundamental traits of Java classes in source code form [7].

Given a JVM class file C, aspa derives $\mathsf{ast}(C)$, an instance (production) of the Class root symbol in the grammar of Fig 2. A Class instance is a tuple with the following labelled attributes: the class type (class); its superclass (superclass); the sets of implemented interfaces, fields, methods, constants, and attributes (as shown); the JVM format version (version); and a flag mask (flags, representing access modifiers like public and other properties [12]).

```
Class ::= class : ObjectType
          superclass : ObjectType
          interfaces : set(ObjectType)
          fields : set(Field)
          methods : set(Method)
          pool : set(Constant)
          attributes : set(Attribute)
          version : i4
          flags : i2
Constant ::= i4 | i8 | f4 | f8 | utf8 | ...
Field ::= name : utf8
          type : Type
          flags : i2
          attributes : set(Attribute)
Method ::= signature : Signature
           flags : i2
           attributes : set(Attribute)
Signature ::= name : utf8
              return : Type
              args : seq(Type)
Attribute ::= name : utf8
              content : AttrType
AttrType ::= Code | Exceptions | ...
Code ::= instructions : seq(Instruction)
         max_stack : u2
         locals : u2
         attributes : set(Attribute)
Instruction ::= aload_0 | iconst_m0 | ...
Type ::= Type[] | ObjectType | int | ...
ObjectType ::= class or interface identifier
i<n> ::= n-byte integer
u<n> ::= n-byte unsigned integer
f<n> ::= n-byte floating point constant
utf8 ::= UTF-8 string
...
```

Figure 2: AST representation of Java classes

Other tuple symbols in the grammar of Fig. 2 have a similar definition to Class, such as Field or Method. For tuple symbols S such as these, attributes k shown in bold identify that two instances of S should only be compared (analyzed for differences) if they have the same value

for k, and are called symbol keys. For instance, two instances of the Method symbol should only be compared if they have the same value for the signature attribute, i.e., two methods are comparable if they have the same signature (same name, return type, argument count and types). Also for tuple symbols, we use notation seq(S) and set(S) for some attributes in correspondence to sequences and sets of instances of symbol S, respectively, for instance instructions : seq(Instruction) in Code and methods : set(Method) in Class. The grammar is completed by terminal symbol derivations, as shown for Java bytecode instructions (Instruction) and constant values (integers, UTF-8 strings, etc).

```
// Old version
package toy;
class Foo {
 private int x;
 public Foo(){ x = 0; }
 public int sqX(){ return x * x; }
 public int getX(){ return x; }
}
// New version
package toy;
class Foo {
 // y added
 private int x, y;
  // sqX moved, but unchanged
 public int sqX(){ return x * x; }
 // constructor changed
 public Foo(){ x = 1; y = 0; }
 // getX removed
 // setX added
 public void setX(int v){ x = v; }
}
```

Figure 3: Toy example — Java source code

The conversion from JVM to AST form abstracts a number of serialization features that are syntactically irrelevant and help generating compact patches, as opposed to being sensitive to the particular layout of a JVM class file. Essentially, aspa factors out two main aspects. First, aspa resolves constant pool index references [12] at the AST level. Constant pool indexes are spread throughout the entire contents of a JVM class file and induce low-level binary changes, when the index of a particular constant changes in-between software versions. Secondly, the definition order of several symbols like fields, methods, etc is also factored out, as determined by the set(S) definitions in Fig. 2. Hence, for instance, aspa will consider two class files to be equivalent if they only differ by the use of different JVM pool indexes for constants, or the order of definition of methods or fields.

Example. We first illustrate the process of patch derivation intuitively using a toy example. Two versions of a class named Foo are shown in Fig. 3 and a human-readable representation of the patch between the two versions is shown in Fig. 4. We omit AST representations of old and new versions of Foo, as they would repeat the source code traits, and are in any case also implicit in the patch representation shown. The changes from old to new version are annotated in Fig. 3 as Java comments and in Fig. 4 by notation $=$, $+$, $-$, and p in correspondence to unchanged, added, removed and patched (i.e., changed) sections of the AST, respectively. The changes from old to new version of Foo are then as follows:

```
p Foo {
 p constants {
   + utf8 "y"
   - utf8 "getX"
   + utf8 "setX"
   ...
 }
 p fields {
   + name=y type=int flags=...
 }
 p methods {
  p Foo() {
   p attributes {
    ...
    p Code {
     p instructions {
      = aload_0
      = invokespecial java/lang/Object()
      = aload_0
      - iconst_0
      + iconst_1
      = putfield x
      + aload_0
      + iconst_0
      + putfield y
      = return
     }
    }
   }
   - int getX()
   + void setX(int) { ... omitted ... }
 }
}
```

Figure 4: Toy example — derived patch

— Field y is added;

— Method getX is removed, method setX is added, while unchanged method sqX is unaccounted for by the patch, in spite of being defined in a different order;

— The Foo constructor method is patched: its JVM bytecode instructions contain a different value to initialize field x (1 in place of 0), plus new instructions to initialize field y;

— Constants are added to or removed from the JVM constant pool in relation to all other changes, as exempli-

fied in Fig 3 for the UTF-8 constants in the p constants section ("y", "getX" and "setX").

Path derivation. As illustrated by the example in Fig. 4, an aspa patch is a type of tree-edit script [5] over the AST representation of a Java class. Given two AST representations, A_O and A_N, aspa matches the structure of A_O and A_N and derives as a result the patch $p = \mathsf{diff}(A_O, A_N)$, such that $A_N = p(A_O)$. Generally, to derive the patch p from s_O to s_N, where s_O and s_N are two instances of some symbol S in the AST grammar, aspa proceeds in syntax-driven manner as follows:

— If s_O and s_N are plain terminals (e.g., instances of Instruction) and $s_O \neq s_N$ then p is expressed (fully) by s_N. If $s_O = s_N$, for this and all the cases below, we define p as the identity mapping (denoted by $=$ in Fig. 4);

— If S is a tuple symbol $S = (a_1 : S_1, \ldots, a_n : S_n)$, for instance Class, then the patch is also a tuple $p = (p_1, \ldots, p_n)$ where $p_i = \mathsf{diff}(s_O(a_i), s_N(a_i))$ for $i = 1, \ldots, n$;

— If s_O and s_N are instances of a set attribute $\mathsf{set}(S)$, such as $\mathsf{methods} : \mathsf{set}(\mathsf{Method})$ in Class, then p can be derived using a set difference analysis that takes into account the key attribute k of S if defined (e.g., signature in Method). Changes, additions and removals from s_O to s_N can be identified in this manner, and "tree moves" (i.e., definition order) can be factored out. Note that changes account for possible elements in both s_O and s_N with the same key value, but which differ in some manner otherwise, e.g., like the Foo constructor patch within the p methods section of Fig. 4;

— Finally, if s_O and s_N are instances of of a sequence attribute $\mathsf{seq}(S)$, such as $\mathsf{instructions} : \mathsf{seq}(\mathsf{Instruction})$ in Code, then p can be expressed as a shortest-edit script (SES) over the longest common subsequence (LCS) of s_O and s_N [4]. The SES expresses a sequence of symbol additions, removals, and changes from s_O to s_N, and is derived by aspa using the LCS/SES algorithm described in [21].

Patch application. Given an AST representation A_O of a Java and a patch $p = \mathsf{diff}(A_O, A_N)$ for some other AST representation A_N, p can be applied to A_O to yield A_N, i.e., $A_N = p(A_O)$. The procedure is symmetrical to that of patch derivation described above, hence we omit details that would be repetitive. It should suffice to say aspa changes A_O in syntax-driven manner, accounting for the incremental changes defined by p, resulting in A_N at the end.

Patch format. The aspa binary patch format uses special marks to denote symbol changes, additions, removals, etc, but otherwise simply adheres to the JVM format to encode the AST representation in binary form. For instance, if aspa encounters a method that has been added to a class, it serializes the method definitions (including JVM bytecode) using the JVM format to the patch file. When reading the same patch file for application, that method will be converted (resolved) from the JVM format back to an AST form. This amounts to (reusing) the same mechanism to convert between entire classes in JVM format and corresponding AST representations.

3 Case-study

The J2SE API. The J2SE API is bundled in the rt.jar JAR archive of the Java Runtime Environment (JRE) distribution by Oracle. The archive contents include well-known J2SE API packages, such as java.lang or java.util. To conduct a case-study analysis, we downloaded all JRE Java 7 versions for Linux x64 and extracted the rt.jar archive from each of them. The versions at stake comprise the initial JRE 7 release, plus all subsequent updates available from Oracle's J2SE homepage as of April 4, 2013: updates 1 to 7, 9 to 11, 13, 15, and 17 (updates 8, 12, 14, and 16 are not made available).

Patch derivation. For each pair of successive JRE 7 releases, we derived aspa patches for rt.jar using the jardiff.sh utility script included in the aspa distribution [13]. This script is able to produce a single patch file, reflecting the differences of all class files between two versions of a JAR file. The derived patch can then be applied to the source version JAR using the jarpatch.sh script [13].

For comparison, we also derived patches for rt.jar using the bsdiff [16, 17] binary patching tool. The tool is a well-known one for this purpose. For instance, bsdiff is embedded in Google's Courgette tool to produce Google Chrome patches [1]. We only refer to the comparison of aspa with bsdiff because it is the binary patching tool we have tested that compares more favorably with aspa . The comparison of aspa with other binary patching tools is reported in [13]. For instance, the table shows that JRE update 1 changed (patched) 20 classes over the initial JRE release, added 3 new ones, and removed none.

We did two adjustments to make the comparison between aspa and bsdiff patches as balanced as possible. First, since bsdiff employs built-in bzip2 -9 compression for patches, we compressed aspa patches in the same manner. Secondly, we ran bsdiff not over the rt.jar archives directly, but over corresponding files containing the concatenation of all JVM class files in the rt.jar archive, ordered by package and class names. The latter aims to factor out too many dependencies of the JAR archive format itself in bsdiff patches, which aspa is capable of dealing with comparable less impact. We examine the sensitivity of aspa and bsdiff patches to variations in the binary input format later in the text.

V	p	+	−	∑	aspa	bsdiff
u01	20	3	0	23	8.2	12.3
u02	91	6	0	97	36.7	67.3
u03	25	3	0	28	27.2	35.4
u04	431	61	3	495	156.1	292.4
u05	46	0	0	46	12.5	27.7
u06	365	17	205	587	108.9	161.1
u07	78	36	8	122	27.1	47.1
u09	55	14	0	69	21.9	32.8
u10	26	5	0	31	13.4	22.1
u11	3	0	0	3	2.3	3.5
u13	150	28	18	196	51.0	94.0
u15	24	2	1	27	13.2	17.6
u17	10	2	0	12	8.9	12.0

V: JRE version for rt.jar; p: patched classes; +: added classes; −: removed classes; ∑: total number of patched/added/removed classes; aspa: size of aspa patch (KB); bsdiff: size of bsdiff patch (KB)

Table 1: rt.jar class changes and patch sizes

Patch size comparison. We summarize in Table 1 the evolution of the rt.jar archive between successive JRE 7 updates, and the sizes of corresponding aspa and bsdiff patches. The numbers shown for patched, added, and removed classes in each JRE 7 update were calculated by aspa during patch derivation.

The general conclusion to draw from Table 1 is that aspa patches can be significantly smaller than bsdiff patches. On average, the size of the derived bsdiff patches was 1.65 times the size of aspa patches, from a minimum factor of 1.3 (for u3 and u15), to a maximum one of 2.2 (for u05).

We also measured the cross-correlation coefficient between the statistical distributions of total class changes (the ∑ column in Table 1) and the size of patches (the aspa and bsdiff columns). The coefficients are 0.94 for aspa and 0.90 for bsdiff , indicating that aspa patches seem to be in more modular correlation to actual variations between versions of the rt.jar archive.

Binary patching sensitivity. The size of binary patches is naturally sensitive to variations in the input format, and can be quite disproportionate to the actual changes between software versions. We illustrate this point with two examples:

1) If, similarly to aspa, the JAR files are provided directly as input to bsdiff (in place of "flat" JVM files, described previously), the patch size is increased significantly. For instance, the bsdiff patch for the update between JRE versions u10 and u011 (the smallest in size in Table 1) grows from 3.5 KB to 79.12 KB. In contrast, aspa abstracts JAR file entry details (like CRC checksums or timestamps) that are irrelevant.

2) The bsdiff patches will even become larger, and by another order of magnitude, if we use an aspa generated JAR in place of the (AST-equivalent) JRE counterpart, given that aspa follows its own JVM serialization strategy when producing JAR/JVM files, in particular reordering the binary order of several definitions (e.g., methods). For the same case as above, the bsdiff patches grow from 3.5 KB to 531 KB, if provided with the JRE u10 JAR file and the u11 JAR file generated by aspa (AST-equivalent to the u11 JRE version, hence inducing the same aspa patch).

Timings. The experiments for our case-study ran on a 2.5 GHz Intel Core i5 machine with 4 GB of memory. We measured the times for aspa and bsdiff patch derivation and application. For aspa we measured an average time of 60.4 seconds (s) for patch derivation and 7.2 s for patch application. As for bsdiff , the average times were 31.0 s for patch derivation and 1.2s for patch application.

The times for patch application are specially relevant, as they will determine the benefit of transmitting compact patches over a network vs. the option of transmitting full software versions. Given that the compressed size of rt.jar can be about 13 MB using bzip2 compression (the JAR files in a JRE distribution contain class files in uncompressed JVM format, since compression is only applied over the entire JRE distribution bundle), a 7.2 s download time (the average time to apply a aspa patch) would be feasible with a 1.8 MB/s (14.4 Mbits/s) download rate.

The observation above signals a concern for subsequent improvement in aspa. The time for patch application reflects the prototype stage of the tool, particularly in regard to I/O implementation details. A great proportion of the time (approx. 85%, 6.1 out of 7.2 s) is consumed by aspa on I/O operations producing the target JAR file, while deriving the AST representation from the source JAR file and changing it through a patch take considerably less longer (approx. the other 15%).

4 Discussion

Summary. We have proposed a methodology for deriving patches for Java software upgrades, based on an AST-level representation of JVM class files, implemented by the aspa software tool. The J2SE API case-study demonstrates the effectiveness and flexibility of the approach: aspa patches were found to be significantly smaller than patches generated by state-of-the-art binary patching tools, and are insensitive to binary-level

changes that do not correlate with actual changes in Java source code.

Other languages and compilation formats. Our proposal may naturally generalize to other languages and compilation formats. For instance, other virtual-machine based languages like C# or Python are in principle quite amenable to our methodology. We can also think of Java again, but considering the Dalvik bytecode format [6] that is used in Android devices. This is a an interesting direction for future work, as Android apps are updated frequently and in full, leading to long upgrade times and bandwidth consumption charges for the end user. A more complex scenario is that of programs compiled onto native code, e.g., C/C++ programs. In this case it may be harder, in principled or technical terms, to derive patches that relate to source code changes in fine-grained manner. Even so, tools like Courgette [1] demonstrate that factoring out some irrelevant (even if low-level) differences in the compilation format can lead to much smaller binary patches.

AST differential analysis. The core trait of our methodology is the use of a high-level AST representation and an associated differential analysis. We are not aware of previous work that considers the derivation of patches for *compiled* programs in AST-driven manner with the specific intent of enabling software upgrades. The AST-difference approach has been employed however for empirical analysis of software evolution [2, 14, 22], a type of application for which we aim to extend `aspa` in the future, or to derive patches for data formats like XML [11]. We also consider that AST-based software patches may in particular provide an appropriate abstraction level for analysis in special and complex contexts, like differential symbolic execution [18] or dynamic software updates (DSU) [3, 15, 20]. For instance, in the case of DSU, patches are applied at runtime and a fine-grained analysis is required over the extent and type of changes to decide if and how a patch can be applied during the execution of a program.

Modular software evolution. In a broader sense, the problem approached by this paper relates to principled and modular change analysis in a software evolution context, in line with past work by the author in this vein [8, 9]. The general underlying concern is to attain principled abstractions for software evolution, such that we can reason on a modular relation between changes to a software artifact and their impact.

Acknowledgements. The author wishes to thank the anonymous reviewers for their comments on the draft version of this paper, and João Sousa for help and encouragement. This work has been partially funded by the Large-Scale Informatics Systems Laboratory (LaSIGE) at Faculdade de Ciências da Universidade de Lisboa.

References

[1] ADAMS, S. Smaller is faster (and safer too), `http://blog.chromium.org/2009/07/smaller-is-faster-and-safer-too.html`, 2009.

[2] APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. JDiff: A differencing technique and tool for object-oriented programs. *Proc. ASE 14*, 1 (2007), 3–36.

[3] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proc. Eurosys* (2009), ACM, pp. 187–198.

[4] BERGROTH, L., HAKONEN, H., AND RAITA, T. A survey of longest common subsequence algorithms. In *Proc'SPIRES* (2000), IEEE, pp. 39–48.

[5] BILLE, P. A survey on tree edit distance and related problems. *Theoretical computer science 337*, 1 (2005), 217–239.

[6] BORNSTEIN, D. Dalvik VM internals. In *Google I/O Developer Conference* (2008), vol. 23, pp. 17–30.

[7] GOSLING, J., JOY, B., STEELE, G., BRACHA, G., AND BUCKLEY, A. *The Java language specification (version 7)*. Oracle Corp., 2013.

[8] KIRSCH, C., LOPES, L., AND MARQUES, E. Semantics-preserving and incremental runtime patching of real-time programs. In *Proc. APRES* (2008), ARTIST Network of Excellence, pp. 3–7.

[9] KIRSCH, C., LOPES, L., MARQUES, E., AND SOKOLOVA, A. Runtime programming through model-preserving, scalable runtime patches. In *Proc. ACSD* (2011), IEEE Computer Society, pp. 77–86.

[10] LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proc. IEEE 68*, 9 (1980), 1060–1076.

[11] LINDHOLM, T., KANGASHARJU, J., AND TARKOMA, S. Fast and simple XML tree differencing by sequence alignment. In *Proc. DocEng* (2006), vol. 10, pp. 75–84.

[12] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine specification*. Addison-Wesley, 1999.

[13] MARQUES, E. `aspa` : a patching tool for JVM class files, `http://www.di.fc.ul.pt/~edrdo/aspa`, 2013.

[14] NEAMTIU, I., FOSTER, J. S., AND HICKS, M. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes 30*, 4 (2005), 1–5.

[15] NEAMTIU, I., AND HICKS, M. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI* (2009), pp. 13–24.

[16] PERCIVAL, C. Binary diff/patch utility, `http://www.daemonology.net/bsdiff`.

[17] PERCIVAL, C. *Matching with mismatches and assorted applications*. PhD thesis, University of Oxford, 2006.

[18] PERSON, S., DWYER, M., ELBAUM, S., AND PSREANU, S. Differential symbolic execution. In *Proc. FSE* (2008), ACM, pp. 226–237.

[19] POTTER, S. Using binary delta compression (BDC) technology to update Windows XP and Windows Server 2003. Microsoft Corp., 2005.

[20] SUBRAMANIAN, S., HICKS, M., AND MCKINLEY, K. Dynamic software updates: A VM-centric approach. In *Proc. PLDI* (2009), ACM, pp. 1–12.

[21] WU, S., MANBER, U., MYERS, G., AND MILLER, W. An O(NP) sequence comparison algorithm. *Information Processing Letters 35*, 6 (1990), 317–323.

[22] XIE, G., CHEN, J., AND NEAMTIU, I. Towards a better understanding of software evolution: An empirical study on open source software. In *Proc. ISCM* (2009), IEEE Computer Society, pp. 51–60.