What *can* Java Binary Compatibility mean?

Sophia Drossopoulou, Susan Eisenbach, David Wragg
Department of Computing, Imperial College,

Abstract

Java *binary compatibility* prescribes conditions under which modification and re-compilation of classes does not necessitate re-compilation of further classes importing the modified classes. Binary compatibility is a novel concept for language design.

We argue that the description of the term binary compatibility in the Java language specification allows for many possible interpretations. We discuss the various interpretations and their ramifications, and suggest one interpretation, which is best in our view.

1 Introduction

Separate compilation and linking were introduced into programming languages in the seventies. In the traditional arrangement, *e.g.* Ada [9], Modula-2 [10, 2], the compiler checks for type consistency and the linker resolves references and checks the order of compilation. Any units (*i.e.* packages, classes, modules) importing modified units have to be re-compiled. So, separate compilation of several units corresponds to the compilation of all units together.

However Java [7], has a different approach, whereby the remit of the linker has been extended: not only does it have to resolve external references, it also has to ensure that binaries (the compiled units) are structurally correct (verification), and that they respect the types of entities they import from other binaries (resolution); however, the order of compilation need not correspond to the import relation. This approach allows for dynamic linking and execution of remotely produced code whose source code is not necessarily accessible.

In particular, certain source code modifications, such as adding a method to a class, are *binary compatible* [6]. The Java language description does not require re-compilation of units importing units modified in binary compatible ways, and claims that successful linking and execution of the altered program is guaranteed. Not only do binary compatible changes not require re-compilation of other units, but such re-compilations *may not be possible*: a binary compatible change to the source code for one class may cause the source code of other classes no longer to be type correct. Separate compilation is *not* equivalent to compilation of all units together.

Because of the related security issues [3], and implications on library modification policies, binary compatibility has practical importance. The concept is rather complex – the language specification is inconsistent in some places, as it considers certain changes to be binary compatible, whose combination can be shown to lead to programs which cannot link [5]. More importantly, the escription given in the language specification allows for several interpretations.

In [4, 5] we developed formalizations to study issues around compilation, linking and binary compatibility. In the process, we discovered four different interpretations of the defintion of binary compatibility given in the Java specification [7] each with different properties.

This paper offers a less formal approach than [4] and concentrates on the issues around the meaning of binary compatibility. In section 2 we introduce Java binary compatibility. In section 3 we introduce basic notions of compilation, linking and compile-time or link-time checks. In section 4 we explore and compare the four interpretations of binary compatibility. Finally, in section 5 we draw conclusions.

2 Binary compatibility in Java

The concept of binary compatibility in Java is motivated by the intention to support large scale re-use of software available on the Internet [8]. Also, binary compatibility aims to avoid the *fragile base class problem*, found in most C++ implementations, where a field (data member or instance variable) access is compiled into an offset from the beginning of the object, fixed at compile-time. If new fields are added and the class is

1st phase

```
class Student { int grade; }
class CStudent extends Student { }
class Lab {
    CStudent guy;
    void f(){ guy.grade=100; }
    }
```

2nd phase

```
class CStudent extends Student {
    char grade;
    }
```

3rd phase

```
class Marker {
    CStudent guy;
    void g(){ guy.grade='A'; }
    }
```

Figure 1: Students and computing students - code

re-compiled, then offsets may change, and object code that previously compiled using the original definition of the class may not execute safely together with the object code of the modified class. Similar problems may arise with virtual function calls.

C++ development environments usually attempt to compensate by automatically re-compiling all files importing the modified class. In Java, although some development environments apply the same strategy, this would be too restrictive in some cases. For instance, if one developed a local program `P`, which imported a library `L1`, the source for `L1` was not available, `L1` imported library `L2`, and `L2` was modified, then re-compilation of `L1` would not be possible. Any further development of `P` would therefore be impossible.

In contrast, Java promises that if the modification to `L2` were binary compatible, then the binaries of the modified `L2`, the original `L1` and the current `P` can be linked without error. This is possible, because Java binaries carry more type information than object code usually does.

The example in figure 1 demonstrates some of the issues connected with binary compatibility. It consists of three phases. In the first phase we create the classes `Student`, `CStudent`, and `Lab`. The class `CStudent` inherits the instance variable `grade` of type `int`. In class `Lab` the field `guy`, of class `CStudent`, is assigned grade `100`. This program is well-formed and compiles producing binary files `Student.class`, `CStudent.class` and `Lab.class`. In the second phase we add the field `grade` of type `char` to class `CStudent`, and re-compile `CStudent`, producing `CStudent'.class`. In the third phase we define a new class, `Marker`. In the body of its method `g()`, we assign the grade `'A'` to `guy`. The class `Marker` is type correct, and thus it can be compiled to produce the file `Marker.class`.

The two changes, *i.e.* the addition of field `grade` in class `CStudent`, and the creation of class `Marker`, are binary compatible changes. So, the corresponding binaries, *i.e.* `Student.class`, `CStudent'.class`, `Lab.class` and `Marker.class`, can safely be linked together.

The sources are *not* type correct any more. An attempt to re-compile the class `Lab` would flag a type error for the assignment `guy.grade=100`, since the expression `guy.grade` now refers to the field in class `CStudent` which is of type `char`. Also, the compiled form of the expression `guy.grade` in the binary `Lab.class` refers to an integer, whereas the compiled form of the same expression in the binary `Marker.class` refers to a character. The two compiled forms exist at the same time, and refer to different fields of a `CStudent` object; *c.f.* figure 3, where `guy[Student].grade` represents the first and `guy[CStudent].grade` represents the second access.

Similar situations can arise for method calls.

3 Fragments

As in [1] and in [4], we consider *fragments* as the basic units participating in compilation and linking. Fragments are collections of classes or interfaces, and they need *not* be self-contained. As we are interested in compilation and linking we distinguish source code fragments from binary fragments, and we use:

- $\mathcal{S}_{\text{java}}$ to indicate the Java *source* language,
- $\mathcal{B}_{\text{java}}$ to indicate the Java *binary*, or *byte-code* level language. $\mathcal{B}_{\text{java}}$ should contain all information necessary for execution and for compilation of importing fragments.

$\mathcal{S}_{\text{java}}$ fragments will be named `S`, `S'`, $\texttt{S}_1$, *etc*, $\mathcal{B}_{\text{java}}$ fragments will be named `B`, `B'`, $\texttt{B}_1$, *etc.* $\mathcal{S}_{\text{java}}$ fragments for the students example are shown in figure 2, and a high level version of $\mathcal{B}_{\text{java}}$ fragments is shown in figure 3. Although byte-code does not look like the code presented in 3, what we show there essentially contains the ame information as available in the byte-code. The difference between the fragments in figure 2 and 3 is, that field accesses in the latter are enriched with information necessary for execution.

The following distills the basic concepts necessary for our description of compilation and linking:

- The mapping $\mathcal{C} : \mathcal{B}_{\text{java}} \times \mathcal{S}_{\text{java}} \longrightarrow \mathcal{B}_{\text{java}}$ represents compilation of $\mathcal{S}_{\text{java}}$ fragments into $\mathcal{B}_{\text{java}}$ fragments using environment information from imported $\mathcal{B}_{\text{java}}$ fragments.
- The operator $^+ : \mathcal{S}_{\text{java}} \times \mathcal{S}_{\text{java}} \longrightarrow \mathcal{S}_{\text{java}} \cup \mathcal{B}_{\text{java}} \times \mathcal{B}_{\text{java}} \longrightarrow \mathcal{B}_{\text{java}}$ combines fragments forming larger fragments.

1st phase

S^{st} = `class Student { int grade; }`

S^{cs} = `class CStudent extends Student { }`

S^{lab} =
```
class Lab {
    CStudent guy;
    void f(){ guy.grade=100; }
}
```

2nd phase

S^{st} = *as in 1st phase*

$S^{cs'}$ =
```
class CStudent extends Student {
    char grade;
}
```

S^{lab} = *as in 1st phase*

3rd phase

S^{st} = *as in 1st and 2nd phase*

$S^{cs'}$ = *as in 2nd phase*

S^{lab} = *as in 1st and 2nd phase*

S^{m} =
```
class Marker {
    CStudent guy;
    void g(){ guy.grade='A'; }
}
```

Figure 2: Computing students - source language fragments

1st phase

B^{st} = `class Student { int grade; }`

B^{cs} = `class CStudent extends Student { }`

B^{lab} =
```
class Lab {
    CStudent guy;
    void f(){ guy[Student].grade=100; }
}
```

2nd phase

B^{st} = *as in 1st phase*

$B^{cs'}$ =
```
class CStudent extends Student {
    char grade;
}
```

B^{lab} = *as in 1st phase*

3rd phase

B^{st} = *as in 1st and 2nd phase*

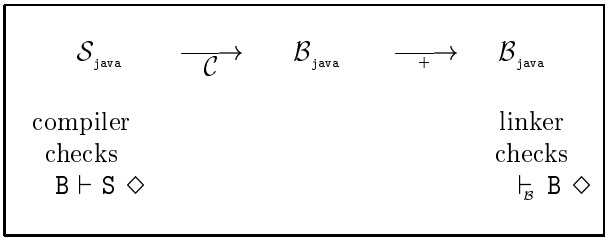
$B^{cs'}$ = *as in 2nd phase*

B^{lab} = *as in 1st and 2nd phase*

B^{m} =
```
class Marker {
    CStudent guy;
    void g(){ guy[CStudent].grade='A'; }
}
```

Figure 3: Computing students - intermediate language fragments

- $\_ \vdash \_ \diamond$ is a relation in $\mathcal{B}_{\text{java}} \times \mathcal{S}_{\text{java}}$ representing compile-time checks. For $\mathcal{S}_{\text{java}}$ fragment S, $\mathcal{B}_{\text{java}}$ fragment B, the assertion $B \vdash S \diamond$ expresses that no errors would be flagged by t when compiling S in the environment of B. $\mathcal{C}\{\!\{B, S\}\!\}$ is defined iff $B \vdash S \diamond$.
- $\vdash_{\mathcal{B}} \_ \diamond$ is a relation in $\mathcal{B}_{\text{java}}$, representing link-time checks. The assertion $\vdash_{\mathcal{B}} B \diamond$ expresses that no errors should be flagged when linking B.



The source code of the first phase of the computing students example consists of $S^{st}+S^{cs}+S^{lab}$. Compilation is represented by $\mathcal{C}\{\!\{\epsilon, S^{st}\}\!\} = B^{st}$, and $\mathcal{C}\{\!\{B^{st}, S^{cs}\}\!\} = B^{cs}$. The intermediate code of the first phase consists of $\mathcal{C}\{\!\{\epsilon, S^{st}+S^{cs}+S^{lab}\}\!\} = B^{st}+B^{cs}+B^{lab}$. In the second phase we compile $\mathcal{C}\{\!\{B^{st}+B^{cs}+B^{lab}, S^{cs'}\}\!\} = B^{cs'}$, thus, the intermediate code of the second phase consists of $B^{st}+B^{cs'}+B^{lab}$.

Note that compiling after linking, *i.e.* $\mathcal{C}\{\!\{B, S_1+S_2\}\!\}$, need *not* be equivalent to linking after compilation, *i.e.* to $\mathcal{C}\{\!\{B, S_1\}\!\}+\mathcal{C}\{\!\{B, S_2\}\!\}$. For example, $\mathcal{C}\{\!\{B^{st}, S^{lab}\}\!\}$ is undefined, whereas $\mathcal{C}\{\!\{B^{st}, S^{lab}+S^{cs}\}\!\} = B^{lab}+B^{cs}$.

Linking intermediate code in actual systems may involve several steps, *e.g.* verification of format, resolution of references, and several checks, often applied in an interleaved manner. We are not interested in these steps themselves, and we consider that all checks should take place when testing well-formedness of the fragment resulting from the linking process. Thus, the case where linking fragments B_1 and B_2 should flag an error can be modelled by $\vdash_{\mathcal{B}} B_1+B_2 \diamond$ *not* holding.

We shall call fragments *disjoint* if they define classes and/or interfaces with different names. For example, $S^{cs}+S^{st}$ and S^{m} are disjoint, whereas $S^{st}+S^{cs}$ and $S^{m}+S^{cs'}$ are not. Fragments "containing" other fragments are said to subsume them. For example, $S^{lab}+S^{st}+S^{cs'}+S^{m}$ subsumes $S^{m}+S^{cs'}$.

Binary compatibility is concerned with the effects of modifications to source and intermediate code. Therefore we define the operator $\oplus$ to describe the effect of *updating* the first argument by the definitions from the second, whereby any entity in both will be taken from the second. For example $((S^{st}+S^{cs})+S^{lab}) \oplus S^{cs'} = S^{st}+S^{cs'}+S^{lab}$. Also, $(B^{st}+B^{lab}) \oplus B^{m} = B^{st}+B^{lab}+B^{m}$.

Thus, the second phase of our example compiles $S^{cs'}$ into $B^{st}+B^{cs}+B^{lab}$, *i.e.* $B^{st}+B^{cs}+B^{lab} \oplus \mathcal{C}\{\!\{(B^{st}+B^{cs}+B^{lab}), S^{cs'}\}\!\}$, giving $B^{st}+B^{cs'}+B^{lab}$.

4 Four interpretations of binary compatibility

The Java language specification [7] describes binary compatible changes as follows:

> "*A change to a type is binary compatible with (equivalently, does not break compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.*"

Thus, binary comatibility restricts source code modifications in terms of properties of the resulting compilation, and its formalization will have the general form:

> S is a *binary compatible change* of B iff:
> $\vdash_{\mathcal{B}} \ldots \mathtt{B} \ldots \diamond \implies \vdash_{\mathcal{B}} \ldots \mathtt{B} \ldots \oplus \mathcal{C}\{\!\{\ldots \mathtt{B} \ldots, \mathtt{S}\}\!\} \ldots \diamond$

During the process of formalization, we realized that the definition from [7] is *not* unambiguous. Namely, it does not make clear how many binaries are meant, and in which environment the compilation of the modified code should take place. This gives rise to several possible interpretations. In the remainder of this paper we explore the four alternatives.

1st interpretation considering one set of binaries:

> "*A change to a type is binary compatible with pre-existing binaries if* these same pre-existing *binaries will continue to link without error if they previously linked without error.*"

which would be formalized as:

Def 1 S *is a* weak binary compatible change *of* B *if*:

$$\vdash_{\mathcal{B}} \mathtt{B} \diamond \quad \implies \quad \vdash_{\mathcal{B}} \mathtt{B} \oplus \mathcal{C}\{\!\{\mathtt{B}, \mathtt{S}\}\!\} \diamond$$

Therefore, $\mathtt{S}^{\mathtt{cs}'}$ is a weak binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$. Still, definition 1 is too weak, as it allows the removal of features not called in the reference binary B, with no regard to further libraries $\mathtt{B}_0$ which linked with B and possibly relied on these features.

For example, $\mathtt{S}^{\mathtt{cs}}$ is a weak binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'}$, effectively removing field `char grade` from class `CStudent`, even though $\mathtt{B}^{\mathtt{lab}}$ linked with $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}'}$, but does not link with $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$.

Therefore, we believe that more than one set of binaries should be meant:

> "*A change to a type is binary compatible with* certain *pre-existing binaries if* any further pre-existing binaries *that previously linked without error* with the pre-existing binaries *will continue to link without error.*"

Still, the question as to the environment of compilation of the modification remains open. This could be the first or the first *and* second set of binaries. This question leads to three further interpretations: strong binary compatible change, binary compatible change, and binary compatible change in context. These interpretations are refinements of each other: if S is a binary compatible change of B in the context $\mathtt{B}_1$, then S is a binary compatible change of $\mathtt{B} + \mathtt{B}_1$; if S is a binary compatible change of B, then there exists a $\mathtt{B}_1$ so that S is a strong binary compatible change of $\mathtt{B} + \mathtt{B}_1$.

Thus, the more refined interpretations require a smaller "reference" binary B. At the end of this section we show why we attatch such importance to small reference binaries B.

2nd interpretation considering two sets of binaries, and compiling in the first set:

> "*A change to a type is binary compatible with* certain *pre-existing binaries if* any further binaries *that linked without error* with the pre-existing binaries *will continue to link without error* with the result of the compilation in the environment of the first binaries."

formalized as:

Def 2 S *is a* strong binary compatible change *of* B*, iff for all* $\mathtt{B}_0$ *disjoint from* S:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond \quad \implies \quad \vdash_{\mathcal{B}} \mathtt{B}_0 + (\mathtt{B} \oplus \mathcal{C}\{\!\{\mathtt{B}, \mathtt{S}\}\!\}) \diamond$$

So, $\mathtt{S}^{\mathtt{lab}}$ is a strong binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$. Notice, that S may be a a strong binary compatible change of a library B, which imports other libraries, and which cannot be used in isolation, *i.e.* $\not\vdash_{\mathcal{B}} \mathtt{B} \diamond$ does not hold. Such a library can only be compiled in the presence of further libraries, represented by the fragment $\mathtt{B}_0$, with which $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond$. Thus, B acts as a filter for $\mathtt{B}_0$, by requiring that $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond$.

Still, definition 2 is too strong, because it expects B to contain *all* information necessary for the compilation of S. Thus, for the following source:

```
S^test  =  class Test
             { CStudent don; ... don.grade = 99; }
```

$\mathtt{S}^{\mathtt{test}}$ is a strong binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$, even though $\mathtt{S}^{\mathtt{test}}$ only *uses*, and does *not modify* features from $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$.

3rd interpretation taking two sets of binaries into account, and compiling in the two sets of binaries:

> "*A change to a type is binary compatible with* certain *pre-existing binaries if* any further binaries *that linked without error* with the pre-existing binaries *will continue to link without*

error with the result of the compilation in the environment of both binaries. "

considers $\mathtt{S}$ to be a *binary compatible change* of $\mathtt{B}$, if all fragments $\mathtt{B}_0$ that successfully linked with $\mathtt{B}$ continue to do so after compilation of $\mathtt{S}$ into $\mathtt{B}_0 + \mathtt{B}$.

Def 3 $\mathtt{S}$ *is a* binary compatible change *of* $\mathtt{B}$*, iff for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond \quad \Longrightarrow \quad \vdash_{\mathcal{B}} (\mathtt{B}_0 + \mathtt{B}) \oplus \mathcal{C}\{\!\{(\mathtt{B}_0 + \mathtt{B}), \mathtt{S}\}\!\} \diamond$$

So, $\mathtt{S}^{\mathtt{lab}}$ is a binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$, and $\mathtt{S}^{\mathtt{cs}'}$ is a binary compatible change of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}} + \mathtt{B}^{\mathtt{lab}}$.

Definition 3 is weaker than definition 2, because it is possible for $(\mathtt{B}_0 + \mathtt{B}) \oplus \mathcal{C}\{\!\{(\mathtt{B}_0 + \mathtt{B}), \mathtt{S}\}\!\}$ to be defined and for $\mathtt{B} \oplus \mathcal{C}\{\!\{\mathtt{B}, \mathtt{S}\}\!\}$ not to be. Thus, $\mathtt{B}$ does not need to contain *all* the type information necessary to compile and type check $\mathtt{S}$; it only needs to contain *enough* information to ensure type correct compilation of $\mathtt{S}$ in the environment of all appropriate fragments $\mathtt{B}_0$, which satisfy $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B} \diamond$.

For example, $\mathtt{S}^{\mathtt{test}}$ is a binary compatible change of $\mathtt{B}^{\mathtt{lab}}$, even though $\vdash_{\mathcal{B}} \mathtt{B}^{\mathtt{lab}} \diamond$ does *not* hold, and even though $\mathtt{B}^{\mathtt{lab}}$ does *not* contain enough information for the compilation of $\mathtt{S}^{\mathtt{test}}$. This is so, because all $\mathtt{B}_0$ which satisfy $\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B}^{\mathtt{lab}} \diamond$ will hold enough features for the compilation $\mathcal{C}\{\!\{(\mathtt{B}_0 + \mathtt{B}^{\mathtt{lab}}), \mathtt{S}^{\mathtt{test}}\}\!\}$.

In previous work [5] we had adopted definition 3. We later realized that the reference binaries $\mathtt{B}$ can be reduced even further. For example, $\mathtt{S}^{\mathtt{cs}'}$ does not modify $\mathtt{B}^{\mathtt{lab}}$, it only uses features used by $\mathtt{B}^{\mathtt{lab}}$; the difference in the roles played by reference binaries is expressed by the concept of a *context*, as in the fourth interpretation.

4th interpretation considering three sets of binaries, and compiling in the three sets of binaries:

"*A change to a type is binary compatible with* certain *pre-existing binaries* in the context of some other binaries *if* any further binaries *that linked without error* with the first and second binaries together *will continue to link without error* with the result of the compilation in the environment of the three sets of binaries. "

Notice, that contexts are not mentioned in [7]; nevertheless, we believe that their advantages justify their introduction:

Def 4 *An* $\mathcal{S}_{\text{java}}$ *fragment* $\mathtt{S}$ *is a* binary compatible change *of an* $\mathcal{B}_{\text{java}}$ *fragment* $\mathtt{B}_1$ *in the context of* $\mathcal{B}_{\text{java}}$ *fragment* $\mathtt{B}_2$ *iff* $\mathtt{B}_2$ *is disjoint from* $\mathtt{S}$, $\mathtt{B}_1$, *and for all* $\mathtt{B}_0$ *disjoint from* $\mathtt{S}$:

$\vdash_{\mathcal{B}} \mathtt{B}_0 + \mathtt{B}_1 + \mathtt{B}_2 \diamond \quad \Longrightarrow$
$\vdash_{\mathcal{B}} (\mathtt{B}_0 + \mathtt{B}_1 + \mathtt{B}_2) \oplus \mathcal{C}\{\!\{(\mathtt{B}_0 + \mathtt{B}_1 + \mathtt{B}_2), \mathtt{S}\}\!\} \diamond$.

So, we distinguish $\mathtt{B}_2$, the *context* which may *not* be modified by the compilation of $\mathtt{S}$, from $\mathtt{B}_1$, which *may*. Thus, $\mathtt{S}^{\mathtt{test}}$ is a binary compatible change of ϵ in the context of $\mathtt{B}^{\mathtt{lab}}$, $\mathtt{S}^{\mathtt{cs}'}$ is a binary compatible change of ϵ in the context of $\mathtt{B}^{\mathtt{st}}$, and a binary compatible change of $\mathtt{B}^{\mathtt{cs}}$in the context of $\mathtt{B}^{\mathtt{st}}$, but is is *not* a binary compatible change of ϵ in the context of $\mathtt{B}^{\mathtt{st}} + \mathtt{B}^{\mathtt{cs}}$.

In [4] we show that $\mathtt{S}$ is a *binary compatible change* of $\mathtt{B}_1$ in the context of $\mathtt{B}_2$, iff $\mathtt{B}_2$ and $\mathtt{B}_1$ are disjoint, and $\mathtt{S}$ is a binary compatible change of $\mathtt{B}_1 + \mathtt{B}_2$. Furthermore, the following lemma, proven in [4], says that two binary compatible changes applicable to disjoint $\mathcal{B}_{\text{java}}$ fragments, can be combined into one binary compatible change in the context of a larger context subsuming the contexts of the original changes.

Lemma 1 *For fragments* $\mathtt{S}_1$, $\mathtt{S}_2$, $\mathtt{B}_1$, $\mathtt{B}_2$, $\mathtt{B}_3$, $\mathtt{B}_4$, $\mathtt{B}_5$, *where* $\mathtt{S}_1$, $\mathtt{S}_2$ *are disjoint,* $\mathtt{B}_1$, $\mathtt{B}_2$ *are disjoint, if:*

- $\mathtt{S}_1$ *binary compatible change of* $\mathtt{B}_1$ *in the context of* $\mathtt{B}_3$*,*
- $\mathtt{S}_2$ *binary compatible change of* $\mathtt{B}_2$ *in the context of* $\mathtt{B}_4$*,*
- $\mathtt{B}_5$ *subsumes* $\mathtt{B}_3$ *and* $\mathtt{B}_4$*,*

then

- $\mathtt{S}_1 + \mathtt{S}_1$ *is a binary compatible change of* $\mathtt{B}_1 + \mathtt{B}_2$ *in the context of* $\mathtt{B}_5$*.*

Thus, we are able to combine binary compatible changes and preserve their linking capabilities, provided that the reference binaries $\mathtt{B}_1$, $\mathtt{B}_2$ are disjoint, even if the corresponding contexts $\mathtt{B}_3$, $\mathtt{B}_4$ are not. Therefore, it is important that the reference binaries (in that case $\mathtt{B}_1$, $\mathtt{B}_2$) are as small as possible. For this reason we believe that definition 4 is the best.

5 Conclusions

Traditionally, rules for binary compatibility would have been learnt by programmers through experience, and would have reflected the behaviour of particular compilers and linkers, rather than being specified by the language semantics. The Java language designers felt that the notion of binary compatibility was something they should explicitly define. However, when we examined their definition, we found counter-examples [5] (*i.e.* a sequence of binary compatible changes that resulted in code that could not be linked).

More importantly, we found that there was a range of alternative interpretations of the definition as given in the language specification. In this paper we have explored this range – although we shall not be surprised

if further alternatives are disovered later. We have suggested that the definition which allows for most combinations of changes whilst preserving linking capabilities should be chosen.

Acknowledgements

We are grateful to David Clarke for feedback, and to Phil Wadler for heated discussions and many useful suggestions.

References

[1] L. Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.

[2] M. Dausmann, S. Drossopoulou, G. Persch, and G. Winterstein. A Separate Compilation System for Ada. In *Proc. GI Tagung: Werkzeuge der Programmiertechnik*. Springer Verlag Lecture Notes in Computer Science, 1981.

[3] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Fourth ACM Conference on Computer and Communication Security*, 1997. Revised version Tech Report number SRI CSL 9704.

[4] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus: Towards a Model of Separate Compilation, Linking and Java Binary Compatibility. Technical Report 99/1, Imperial College Department of Computing, January 1999. available at http://www-dse.doc.ic.ac.uk/projects/slurp/.

[5] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What *is* Java Binary Compatibility? In *OOPSLA*, 1998.

[6] Ira Forman, Michael Conner, Scott Danforth, and Larry Raper. Release-to-Release Binary Compatibility in SOM. In *OOPSLA'95 Proceedings*, 1995.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.

[8] James Gosling and H. McGilton. The Java Language Environment A White Paper, http://java.sun.com/docs/white/langenv, 1996.

[9] US Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815 A.

[10] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.