STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.04

DETECTING BINARY INCOMPATIBLE SOFTWARE COMPONENTS USING DYNAMIC LOADER

ÁRON BARÁTH AND ZOLTÁN PORKOLÁB

Abstract. Modern programming languages support modular development dividing the system into separate translation units and compile them individually. A linker is used then to assemble together these units either statically or dynamically. This process, however, introduces implicit dependences between the translation units. When one or more units are modified in inconsistent way binary incompatibility occurs and may result in unexpected program behavior. Current mainstream programming languages neither specify what are the binary compatibility rules nor provide tools to check them.

In this paper we discuss the details of various cases of binary incompatibility. We implemented a prototype solution in the Welltype programming language to detect binary compatibility by dynamic loader.

1. Introduction

Most of the modern programming languages provide some way for modular development. Program code is usually written into separate source files and compiled individually. These are so called *translation units* [1] then organized into higher abstraction packages, modules or libraries. The separation level of the compilation of these translation units are vary in different programming languages. The Java language [2] requires the proper setting of the `CLASSPATH` environment variable to make connection between translation units. In C and C++ languages [3, 4] usually the header files included to multiple translation units provide the consistency.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: PROGRAMMING LANGUAGES – *Languages Constructs and Features*;

Key words and phrases. programming language, linking, binary compatibility.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

To create an executable code, the individually compiled translation units are assembled into an executable program. For classical programming languages like C, C++, Fortran, etc. where the compilation step results machine specific binary code, some kind of linker [5, 6] connects the translated units. This can happen either *statically*, where the assembled units form a unified entity, or *dynamically*, when the necessary code is collected only in run-time. Modern software systems tend to use the dynamic approach [7, 8] as it results smaller binary code and faster compile/link time.

For programming languages using some virtual execution environment, e.g. the virtual machine in Java, the run-time environment provides the proper connection between the units.

There are a number of advantages of this code organization: programmers can work on individual source files with minimal interference. Libraries created from a set of translation unit form reusable subsystems. Compilers can better localize the possible issues when translate the source code. On incremental development only the modified code should be recompiled thus the development time is shorter.

Although the translation units are compiled individually, in many cases there are implicit dependences between them. One unit can use variables or functions defined in some other unit. Objects are defined in one unit as instances of types defined in an other unit. When one or more components are changed most programming languages require full recompilation of the system to ensure the complete consistency between the units. In practice, however, the full recompilation of the system is rarely the case.

This paper is organized as follows. In Section 2 we overview how the current mainstream languages support binary compatibility. In Section 3 we describe a typical industrial scenario to point to the importance of the binary compatibility and its verifiability. In Section 4 we introduce our prototype solution for the problem in the Welltype experimental programming language. We evaluate this approach in Section 5. We briefly discuss our future plans in Section 6. Our paper concludes in Section 7.

2. Related work

Binary compatibility is an issue poorly recognized by language designers, but can cause serious headache for maintainers of large software projects. When already compiled clients are linked against different versions of libraries, incompatible library versions can cause the client code to crash or even worse, to running in undefined way. This problem frequently occurs with C/C++ programs using dynamic libraries, but the issue is not limited to C++, also happens in Java and other languages. Welltype deeply validates modules to link and forbids incompatible usage.

A classical solution to create binary compatible versions for classes is using the handle-body programming pattern [9]. In C++ this is frequently called as the PIMLP pattern and implemented as a single private data member – a smart or raw pointer – referring to an implementation class written in a separate translation unit. As the evolution of the class is reflected in changes only of the implementation, the object layout of the original class used by the clients never changes. On the negative side of this solution we usually have to allocate the implementation class on the heap which may result run-time overhead.

In C and C++ the GNU compiler team developed a solution [10, 11, 12] to append version number to symbols in the ELF (Executable and Linkable Format) [13, 14] files. These informations later can be used by the static or dynamic linker. This solution might be useful to detect some sort of binary incompatible components.

Even the Java programmers must be aware binary compatibility, although, the Java language is not known about program crashes due to binary incompatible components. The binary compatibility has an own chapter in the *Java Language Specification* [15], suggesting the importance of this topic. The chapter detailing what will produce a binary compatible output, and what are the traps. To understand the importance of binary compatibility in Java programs, we must take a closer look to the problems caused by library upgrades [16]. Another example – which is related to library upgrades – is when a refactoring is made [17].

Apart Welltype, other languages were developed to be aware of binary compatibility. ZL is a C++ compatible language in which high-level constructs, such as classes are defined using macros over a C-like core language [18]. This approach makes many parts of the language easily customizable, e.g. the programmer can have complete control over the memory layout of objects. Using this capability, one can develop binary compatible new versions of ZL language objects.

3. Motivation

Suppose, we have a large software system, implemented in an object-oriented programming language, like C++. Here, many of the subcomponents of the system, like networking, logging, database connections, etc. are implemented as classes or a group of classes placed into libraries. Each of these subcomponents have their own maintenance cycle: they evolves implementing new features, are changed due to bug fixes or performance improvements. If the system is large enough, it is not realistic to recompile the whole system when one or more subsystems have changed.

In the industry a typical solution is the following. Each subsystem is implemented in separate translation units and compiled into *dynamically loadable libraries* (e.g. DLLs in Windows, shared objects in UNIX systems). The public *interface* of the subsystems are exposed in *header files*. Applications are using this common header files via the `#include` preprocessor directive. The applications also using the *implementation* of the subsystems picking the corresponding dynamically loadable libraries in run-time when the application starts.

This scenario allows a relatively good opportunity to maintain even large systems. When any of the subsystems requires changes for maintenance purposes it is enough to change and recompile the one in question. Replacing the old `.DLL` or `.so` with the new version the changes will be enabled for the applications on their next start. However, this upgrade scenario does not allows changes on the interface of the subsystems. Any time the public interface of a subsystem changes, applications using it should be recompiled.

Unfortunately, not changing the public interface does not guarantee the binary compatibility of the system components. The C++ programming language uses *value semantic*, i.e. objects are mapped into bytes in memory directly instead of being represented by some reference which points to heap allocated memory (like Java does it). Every time we declare a variable of a type we allocate the corresponding number of bytes. Client code using that object is directly compiled to utilize *size*, and *offsets* corresponding the object's known layout. Changing the layout, e.g. adding a new (non static) member to a class or changing existing ones brakes these assumptions. To avoid inconsistency between the object's actual layout and the layout known by the already compiled clients we should apply only *binary compatible* changes.

What is a binary compatible change is very hard to decide. Language specifications, like the ISO C++ standard [19], do not even mention binary compatibility. Subtle changes, like making an existing member function virtual or adding a new exception may brake compatibility. Experts are collecting traps and pitfalls [20, 21, 22], but those are specific to platform, compiler or even compiler flags to set optimization level. At the moment there are no reliable tool or method to check whether a new library is binary compatible with its previous version.

Nowadays, the programs are stored in well-known binary formats, and the used format varies through operating system. For example, modern Unix and Linux systems use the ELF [13, 14] format (Executable and Linkable Format) since the nineties. Windows systems use the PE [23] format (Portable

Executable). The common in these formats that they provide symbol sharing across dynamic libraries and other programs. The mechanism is unfortunately is too simple to provide possibility to detect binary incompatibility. The C-style linkage consists of basically just function names without any additional information. The C++-style linkage, however, uses mangled names which encapsulates additional information about the function: encodes the type names of the arguments, including `namespace`s and templates.

4. Welltype Dynamic Loader

As we seen above, binary compatibility is really an issue in long-term development. The incompatibilities can cause the program to crash or, even worse, miscalculations that break invariants. Thus, programmers must take actions to avoid such incompatibilities. While in the current, mainstream programming languages only conventions can get rid of binary incompatibilities, the Welltype language explicitly and strictly specifies the binary compatibility.

The Welltype language [24, 25, 26] is an imperative programming language, designed to be safe: strict syntax and strong type system. While the Welltype language is safe, it is still feature rich – supports algebraic data types. Welltype programs can be dynamically linked, performed by the dynamic loader.

The Welltype Dynamic Loader will validate whether the program to be loaded meets the already loaded restrictions. If the loader finds a program is binary incompatible, then it will be refused from loading.

In order for the dynamic loader to be capable to make decision like that, programs must define their public interface: what elements they require and what elements they publish into the environment. These *elements* can be types, functions, operators, etc. Note that this is a notable difference from the standard ELF or PE formats, where only the function symbol names are stored, and everything else is assumed. However, this applies mainly to the C language, because the C++ symbols contain more information, even type names, since it is required to resolve function overloads.

The Welltype specification states that all external references must be explicitly indicated, the source code must contain what elements needs to be *imported*. These references are used twice: First, during the compilation to let the compiler know what undefined but declared elements can be used. Second, during the dynamic linking to bind the externals to the program. After all externals are bound, the program can be executed. Naturally, the indicated externals will be part of the compiled program if they actually referenced – the unreferenced externals will be ignored.

On the other hand, any program can *export* a set of elements into the runtime environment. These exported elements later can be used by other programs. The *one definition rule* [27] – which is a basic concept of the Well-type language – cannot be violated when exporting functions into the runtime environment. This rule will specify a loading order among the programs, since no program can actually import an undefined element.

The importance of the one-definition rule can be easily understood. It is trivial to define two functions with the same signature while their implementation is different. In order to use the same implementation by all the loaded programs, it is mandatory to enforce this rule. Note that, however, it is not required to load *one* specific implementation into the environment. This dynamic mechanism allows to load different implementations to the same program – while the signatures are match.

The loading procedure will process all export and import sections, in the order specified in the program binary. The algorithm used to export elements is the following:

```
procedure export_element(environ, program, elem)
  if not element_is_exported(environ, elem) then
    if element_is_function(elem) then
      error: one definition rule violated
    end if
  end if
  rep := build_representation(program, elem)
  register(environ, rep)
end procedure
```

The algorithm used to import elements is a bit more complicated but straightforward:

```
procedure import_element(environ, program, elem)
  rep := find_element_by_primary_attr(environ, elem)
  if rep is NIL then
    error: unresolved external
  end if
  if not element_is_compatible(elem, rep) then
    error: element is incompatible
  end if
  write_import_info(program, elem, rep)
  add_reference(program, rep)
end procedure
```

The mentioned *primary attributes* are unique to all elements described in Section 4.1.

4.1. Elements in the binary interface.

The concept of the binary interface consists of the following elements: *function signatures*, *operators*, *exceptions*, and the *types*.

The mentioned **function signature** – as expected – take part in the binary interface. The function signature used in the binary consists of

- the name of the function, which is almost a custom zero-terminated string with lesser exceptions;
- the number of argument;
- the exact types of all the arguments;
- the number of return values;
- the exact types of all return values;
- the `pure` attribute
- and (in case of export) the address of the entry point of the function.

If the signatures match, the loader assumes they functions are binary compatible. The current version of the Welltype language does not specify other attributes to identify a function. This specification might looks inadequate, but the strength of the dynamic loader (and the Welltype language itself) are the types.

Primary attributes of the function signature are: name, number of argument, and the types of the arguments. This is similar to the C++ language, because this is the minimal information required to resolve function overloads.

The mentioned *exact types* in the listing above refers to the in-depth type matching. The binary interface must hold the specification of all types that are involved in export or import mechanism. This applies recursively to other types as well. This topic will be discussed later.

In addition to the already mentioned elements, **operators** are also take part in the binary interface. The Welltype specification aimed the goal to load programs that ,,speak the same interface". Therefore, the used operators are also matched, while it is somewhat unnecessary. The reason behind this design decision is that two expressions are not the same if the operator precedences are not compatible. For example, the expression `a *+ b *- c` can be interpreted two ways depending on the precedences:

(1) `(a *+ b) *- c`
(2) `a *+ (b *- c)`

Moreover, the associativity of the operator is also important, because the expression `a *+ b *+ c` also can be interpreted two ways:

(1) `(a *+ b) *+ c`
(2) `a *+ (b *+ c)`

Therefore, the different precedence and associativity cause the source code to be not the same with different settings. Thus the operator specification consists of:

- operator symbol (we distinguish two varieties: the *classic* operator form which consists of one or more operator symbol character with lesser exceptions; and the *identifier* form which is any identifier that accepted by the parser) – this is the only primary attribute;
- precedence ranging from 3 to 15;
- and the associativity (left or right).

The built-in operator set is fixed in the Welltype specification. Multi-defined operators are also ignored. It is done because if all the three attributes match, it defines exactly the same operator, and will be turn into a simple import.

The **exceptions** are also part of the binary interface. A Welltype program can raise only exceptions that are defined in the program or imported into the program. Note that the declared but not exported exceptions will be implicitly exported, because the program needs a global exception identifier (that globals to the runtime environment). This mechanism will not cause any problems, because exceptions can be exported multiple times while not violating the one-definition rule. Technically, the duplicate exception exports are ignored, and the program will implicitly import it. This can be done because the only informations about an exception is its name. Furthermore, the implicit export is used to make reference to the original program that actually exported the exception. Using the method, all program will know which exception to be raised, and which exception to be caught. Since the exceptions are identified in the binary only by its name, the name of the exception is the primary attribute.

4.2. **Types in the binary interface.** In this section we discuss all the types in details that are part of the binary interface.

Types that specified in the current version of the Welltype language are: enumeration type, function type, data type, record, private record and limited record. The primary attribute is common to all types listed here: the name of the type.

The **enumeration type** consists of:

- name of the type;
- number of enumerators;
- identifier of all the enumerators.

Exact match of the enumeration type is important because the programs will communicate with enumerator indices, and every index must refer to the same enumerator. Also, code might be generated on the imported side. Therefore, the number and the identifier of the enumerators must match.

The primary attribute of the **function type** is only its name, because the Welltype specification identifies all types by only their name. This is a little bit contrary to the specification of the function signature, but the *function type* is a type, not an actual function. However, the function type consists the same attributes as the function signature:

- name of the type;
- the number of argument;
- the exact types of all the arguments;
- the number of return values;
- the exact types of all return values;
- and the `pure` attribute.

In order to import a function type, all attributes listed above must exactly match.

The **data type** is the algebraic data type implementation in Welltype. Therefore, the binary representation must reflect the complexity of this type. The following attributes are stored:

- name of the type;
- number of constructors;
- index of the default constructor;
- for each constructor:
  - name of the constructor;
  - number of types in the constructor;
  - list of the types.

The reason why the index of the default constructor is included in the binary representation is similar to the explanation to the operators. This attribute is somewhat unnecessary, but using different default constructor can result totally different program from the same source code. Moreover, if the actual data type does not have a default constructor, then an important attribute will change. Without default constructor the type is not *default constructible*, and this recursively affect other types. The dynamic loader takes actions to avoid to alter such important attribute like the *default constructible*.

The **record** type is quite similar to the `struct` used in C programming language with one major difference: the complete layout is stored in the compiled binary, and validated by the dynamic loader. The stored layout consists of:

- the name of the type;
- the number of fields the record has;
- the list of the field types;
- and the names of all the fields.

The number of the fields and the types of all the fields are very important, because code will be generated on the importer side, which is specific to the actual layout. For example, the size of the whole record depends on its fields. The records passed through different programs must use the same layout, otherwise crash or miscalculation will occur. The field names are stored only in order to ensure that the records are the same. Example to the importance of the field names can be seen in Figure 1. The layout is definitely the same (two `int` in both cases), but with entirely different semantics. For example, accessing the `y` field will use different memory slots. Therefore, the two records are binary incompatible, but this scenario can be detected only if the record fields are stored.

```
record vec
{
  int x;
  int y;
}
/* or */
record vec
{
  int y; // NOTE: the fields are
  int x; //       swapped
}
```

FIGURE 1. Example code breaking the binary compatibility.

```
record my_record
{
  int  first;
  bool second;
  long third;
}
```

FIGURE 2. Example record to demonstrate the serialization.

For example, the record can be seen in Figure 2 will be compiled into the following sequence:

`"my_record", 3, 2, 1, 4, "first", "second", "third"`

Where the type indices are `int`=2, `bool`=1 and `long`=4.

The record type might not the best choice for all situations, because the record may evolve, or the representation intended to be hidden. Since the record type matches all fields, and the fields are free to access, this construct is not optimal to these purposes. The **private record** is used instead. Only the name of the type is stored (hence it is *,,private"*), and the representation is entirely hidden. For the private record type additional functions are required to be imported (or additional functions must be exported). Because the fields are unknown, thus a constructor function is required to construct them. Note that all private record are still *default constructible* despite of the representation is unknown.

The *non-default constructible* version of the private record is the **limited record**. The semantics is the same as the private record, but the constructor function is not imported/exported. Apart from the *default constructible* attribute, the limited record and the private record are the same construction.

5. Evaluation

Although the binary interface is quite strict, the mechanism is actually usable. A few large (over 20k sloc) Welltype programs are written that highly uses the dynamic linking feature: this provides an ability to these programs to replace the back-end implementation. With the help of the opaque types (`private record` and `limited record`) the details can be hidden, and the back-end can be reduced to a simple API (Application Programming Interface) instead of a over-complicated and embedded implementation. Thus, this organization makes the back-end implementation replaceable, not least easy to understand.

Also, this supposed to be a motivating force to programmers design a compact and clear API. Moreover, this approach forces not to leak implementation details, which in most of the cases is absolutely unnecessary. In C++, the needlessly leaked implementation details are considered as *bad practice*, in Welltype, however, they considered as *never do that.*

6. Future work

The current version of the Welltype language does not support classes. However, introducing the class construction brings issues into the strict syntax, and the forced binary compatibility. As we seen, the Java language has problems with binary compatibility. Thus, this language construction required to be carefully designed to suit into the strict syntax, semantics and binary interface.

7. Conclusion

Binary compatibility is a serious but often underestimated issue in modern programming languages. Current mainstream programming languages neither specify nor provide tools to solve the problem. In this paper we discussed the problem in details and suggested a set of rule to check avoiding inconsistencies between binary components. The Welltype experimental programming language is defined to avoid various traps and pitfalls of the current mainstream languages. One of the improvements of Welltype is the application of the dynamic loader with the capability to detect the possible binary incompatible modules. We implemented a prototype tool-chain of Welltype. Practical experiments show that the rules detecting the binary incompatibility in Welltype are strict enough to filter out critical issues, but still allow maintenance of evolving individual subsystems as binary components.

References

[1] ISO, "ISO/IEC 9899:TC3 – committee draft of the C99 standard – section 5.1.1.1." `http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf`.

[2] J. Bloch, *Effective Java: A Programming Language Guide. The Java Series (2nd ed.)*. Addison-Wesley, 2008.

[3] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[4] B. Stroustrup, *The C++ programming language, 4th Edition*. Addison-Wesley, 2013.

[5] S. Chamberlain and I. L. Taylor, "Using LD the GNU linker," 2010. `http://lib.hpu.edu.vn/handle/123456789/21416`.

[6] "LLD – the LLVM linker," 2018. `https://lld.llvm.org/`.

[7] M. Franz, "Dynamic linking of software components," *Computer*, vol. 30, no. 3, pp. 74–81, 1997.

[8] S. Drossopoulou, G. Lagorio, and S. Eisenbach, "Flexible models for dynamic linking," in *European Symposium on Programming*, pp. 38–53, Springer, 2003.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] "LD VERSION command." `https://sourceware.org/binutils/docs/ld/VERSION.html`.

[11] "ELF symbol versioning with glibc 2.1 and later," 1999. `https://lists.debian.org/lsb-spec/1999/12/msg00017.html`.

[12] M. Stevanovic, *Dynamic Libraries Versioning*, pp. 187–231. Berkeley, CA: Apress, 2014. `https://doi.org/10.1007/978-1-4302-6668-6_10`.

[13] Tool Interface Standard, *Executable and Linking Format (ELF) Specification, Version 1.2*, May 1995. `http://refspecs.linuxbase.org/elf/elf.pdf`.

[14] Santa Cruz Operation, *System V Application Binary Interface*, March 1997. `http://www.sco.com/developers/devspecs/gabi41.pdf`.

[15] Oracle, *Java Language Specification, Chapter 13. Binary Compatibility*, 2018. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html`.

[16] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pp. 64–73, IEEE, 2014. `https://doi.org/10.1109/CSMR-WCRE.2014.6747226`.

[17] I. Savga, M. Rudolf, and S. Goetz, "Comeback!: a refactoring-based tool for binary-compatible framework upgrade," in *Companion of the 30th international conference on Software engineering*, pp. 941–942, ACM, 2008. `https://doi.org/10.1145/1370175.1370198`.

[18] K. Atkinson, M. Flatt, and G. Lindstrom, "ABI compatibility through a customizable language," in *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010* (E. Visser and J. Järvi, eds.), pp. 147–156, ACM, 2010. `http://doi.acm.org/10.1145/1868294.1868316`.

[19] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`.

[20] A. Koenig, "The nightmare of binary compatibility," in *Dr.Dobb's*, 2014. `http://www.drdobbs.com/cpp/the-nightmare-of-binary-compatibility/240166914`.

[21] "How to design a C++ API for binary compatible extensibility," 2010. `https://stackoverflow.com/questions/1774911/how-to-design-a-c-api-for-binary-compatible-extensibility`.

[22] Microsoft, "C++ binary compatibility between Visual Studio 2015 and Visual Studio 2017," 2017. `https://docs.microsoft.com/en-us/cpp/porting/binary-compat-2015-2017`.

[23] M. Pietrek, "Peering inside the PE: a tour of the win32 (R) portable executable file format," *Microsoft Systems Journal-US Edition*, pp. 15–38, 1994.

[24] Á. Baráth and Z. Porkoláb, "Welltype: Language elements for multiparadigm programming," in *Position Papers of the 2017 Federated Conference on Computer Science and Information Systems*, pp. 91–101, 2017. `http://dx.doi.org/10.15439/2017F546`.

[25] Á. Baráth, "Welltype legacy web page." `http://baratharon.web.elte.hu/welltype`, 2014.

[26] Á. Baráth, "Welltype project web page." `http://repo.hu/projects/welltype`, 2018.

[27] ISO/IEC, "ISO/IEC 14882:2003(E): Programming Languages - C++ §3.2 One definition rule [basic.def.odr]," 2003. `http://eel.is/c++draft/basic.def.odr`.

Department of Programming Languages and Compilers, Eötvös Loránd University

Email address: `[baratharon,gsd]@caesar.elte.hu`