Issues with Annotation-based Compile-time Metaprogramming

Anonymous Author(s)

ABSTRACT

Metaprogramming can be made by annotations in the source code linked to metacode that change the compilation process. At compile-time, the metacode can add code, do additional checks, support new syntax, and change the semantics of the language elements. The unrestrained power of metaprogramming damages several expected software features such as code readability, modularity, manutenability, and even determinism. This paper lists several issues with compile-time metaprogramming made with source code annotations. A few of the issues were known and have been stated before. Most of them were intuitively known but they have not been precisely described and their consequences have not been given. The article concludes with suggestions on how to work around the solvable issues.

CCS CONCEPTS

• **Software and its engineering** → **Object oriented languages**; **Data types and structures**; **Domain specific languages**; **Extensible languages**; *Source code generation*; *Macro languages.*

KEYWORDS

Object-oriented programming, Metaprogramming, Metaobjects, Compilers, Software Engineering

ACM Reference Format:

Anonymous Author(s). 2023. Issues with Annotation-based Compile-time Metaprogramming. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Metaprogramming is the programming that occurs when a program handles another one. A *metaprogram* handles a *base program* which can be changed, checked, analyzed, or transformed into another program. The semantics of the language can be modified and new syntax can be introduced. The metaprogram can be a snippet of the base program itself and the handling can occur at pre-processing time, compile-time, or runtime. Metaprogramming is a broad area [2] [9] with many different subareas that employ a myriad of terminologies. In this paper, we are interested only in compile-time metaprogramming (CTMP) made by annotations in the base program.

As the name implies, CTMP occurs during the compilation of a base program and, therefore, demands the collaboration of the compiler. Annotations in the base program are syntactic elements linked to *metacode*, which in this article are considered as snippets of the metaprogram. They can be thought of as functions, subroutines, or procedures of the metaprogram. CTMP with annotations requires collaboration among the compiler, the program, and the metaprogram (composed of metacode). A *protocol* describes how these elements collaborate which is, in some languages [6], called Metaobject Protocol (MOP).

```
@immutable
class Employee {
    ...
    @positiveInt
    private int age;
    private @NonNull Company company;
}
```

Listing 1: An example with annotations

This article reports some problems with compile-time metaprogramming made with annotations. A *problem* is either intrinsic to the area and, therefore, not completely soluble, or linked to the metaprogramming support by current languages. There is a blurred borderline between bad metaprogramming (developers' fault) and metaprogramming issues (language fault). We consider that a language characteristic is a *problem* if it makes it difficult to create correct metaprograms. That borders bad programming because a careful developer would produce correct code even in the presence of the issue. But maybe at a high cost in time and producing fragile code.[1]

The metaprogramming *protocol* is defined by a *language* and encompasses the compiler behavior. Hence, the language also specifies how the compiler interacts with the program and metacode. *Metaprogramming system* is generally used, instead of *language*, when the text emphasizes the *compiler* part of the combination *language-compiler*. This article is organized as follows. The issues with metaprogramming are presented in section 2. Section 3 shows some possible solutions.

2 ISSUES WITH COMPILE-TIME METAPROGRAMMING

This section assumes that the background is class-based object-oriented languages in which metacode can change the compilation process. In the examples, given in a Java-like language, metacode are linked to syntactic elements of the source code[2] such as classes, methods, fields, variables, and statements through the use of *annotations* that start with `@`. Listing 1 shows annotation `immutable` attached to the class and annotations `positiveInt` and `NonNull` attached to fields. Figure 1 illustrates the relationships among the compiler, the metacode associated with annotation `immutable`, and

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Code that becomes invalid at the slight change in the base program.

[2] *Source code* is synonymous to *source file*, a single file with one or more classes.

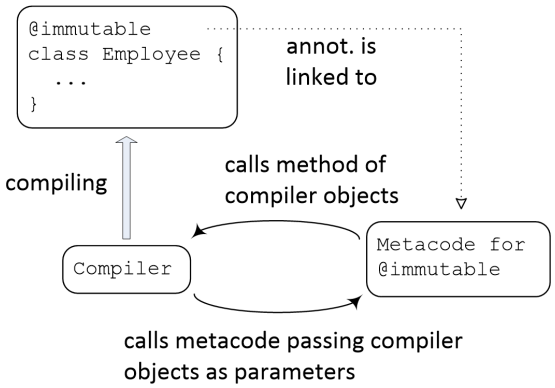


Figure 1: Relations between an annotation, its metacode, and the compiler

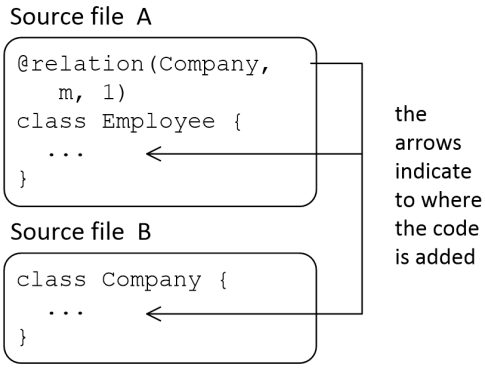


Figure 2: File A changes files A and B

the source code. The metacode for `immutable` is called by the compiler when it finds the annotation. The metacode can call methods of compiler objects. The list of issues with CTMP follows.

MessWithOthers A metacode associated with a source file changes another source file, which is called *obliviousness* [1]. An example is shown in Figure 2 in which annotation `relation` is attached to class `Employee`. This annotation is linked to a metacode that, at compile-time, adds fields and methods to both classes `Employee` and `Company` to implement a many-one relation between them.

From now on, we will use the annotation name as the metacode name. Class `Employee` is in file A and `Company` in file B. Hence, a metacode linked to file A changes another file, B. That breaks modularity because, to understand a source file, the developer has to understand potentially all source files in the program. It is not enough to read the documentation of class `Company` and its source code to know which methods it supplies.

A light version of this problem happens even inside a class because a metacode associated with a method could change another

```
@toJSON(Employee, Company)
class JSONMaster { /* empty */ }
```

Listing 2: toJSON generates methods like `employeeToJSON` for converting an `Employee` object into JSON code

method; add statements to it, for example. Non-local changes like those described make it hard to understand the code.

WhoDependsOnWho The compiler of an object-oriented language typically builds a *class dependence graph* representing the relations between its classes. To make the explanation simpler, suppose there is a one-to-one correspondence between source files and classes. In this graph, vertices are classes and there is an edge from `R` to `S` if `S` has to be recompiled whenever `R` changes. This is the case if `S` inherits from `R` or declares a variable whose type is `R`.

Metacode have to be taken into account to build the *class dependence graph*. Whenever a metacode associated with class `S` uses information about class `R`, there should be an edge from `R` to `S`. This cannot be done if metacode acts in the compiler data structures directly, as when an AST node is passed to a metacode function or method. The handling of the AST node by the metacode is unknown to the compiler and, therefore, it cannot build a *class dependence graph* based on it. An example will be described using the code of Listing 2. The metaobject associated with annotation `toJSON`, attached to the `JSONMaster` class, creates at compile-time a method for each of the annotation parameters, which must be class names. For the first parameter, the method created is

```
String employeeToJSON(Employee obj) { ... }
```

The metaobject uses the fields of `Employee` to generate the method code. Information on the fields is usually got from the AST of the class. After passing the `Employee` AST to metaobject `toJSON`, the compiler generally does not know which AST methods were called. Hence, the metaobject uses information on this class without the knowledge of the compiler and there will be no edge from `Employee` to `JSONMaster` in the *class dependence graph*. The consequences are that, if the former class is changed, the latter will be outdated.

KnowsFriendsSecrets A metacode associated with a class `S` may generate code or do checks based on private information of class `R` as its list of fields, its list of private methods, or even statements of its methods. The use of private information from other source files destroys modularity because class `S` cannot be understood without the knowledge of private information of `R`. This issue with metaprogramming occurs with metaobject `toJSON` of Listing 2 because `JSONMaster` uses fields of `Employee`, which may be private to the class. Knowing private information of objects at runtime, through runtime reflection, has its own problems. However, it is not as bad as having this knowledge at compile time because modularity is not destroyed.

Compiler-Interactions A metacode interacts with the compiler using low-level compiler data structures, like the AST, in several compilation phases. This approach has several drawbacks [17]:

(1) it demands a deep knowledge of the design and implementation of the compiler, which includes details of all the

compilation phases and the data structures used. The metacode may require complex AST transformations that should keep compiler invariants (often undocumented);
(2) incorrect AST handling may crash the compiler or make it generate incorrect code;
(3) metacode may bypass compiler checks causing the acceptance of flawed source code. That is, metacode may add code after the compiler does some checks that will never be done in the added code.

Moreover, metacode become tied to the compiler data structures. Changes to these data structures, like the AST classes, invalidate metaprograms.

This issue happens in languages supporting *compiler plugins* such as [18], Java [12], X10 [11], Kotlin [8], TypeScript [23], and Rust [15]. A *compiler plugin* is composed of metacode called in specific phases of the compilation that interact with the compiler low-level structures. An example of this interaction is that, using the Java plugin, the metacode can set the body of a method (its set of statements) to `null`, crashing the compiler. The developer has to deeply know the compiler to design the metacode because there is no simplified API for compiler plugins, the whole compiler is offered.

The problems cited above happen with any badly designed API and, therefore, one could argue that they are not issues with metaprogramming per si but with many implementations of it. This is correct for many of the issues cited in this paper and clearly visible in Compiler-Interactions. The reason is that, if a problem has a solution (however expensive to implement it is) then it is not related to metaprogramming but to a specific implementation. But this is the usual way of referencing problems in the literature, they are problems only with our current technology.

WhoDidWhat A metacode that handles the compiler data structures directly leaves no traces of its activities. Therefore, if a metacode generates invalid code, detected in later compilation phases, the compiler will issue an error. But it will be unable to point out the metacode that generated the invalid code.

As an example, suppose a Java compiler plugin linked to annotation `getset` inserts a code, before semantic analysis, with a semantic error:

```
void getAge() { return age; }
```

This is done by changing the compiler data structures directly and, therefore, the compiler is unaware of it. The semantic error will be later detected but there will be no indication of which plugin caused the error. Even identifying the plugin is not enough because the source code may have several annotations, like `getset`, linked to the same plugin. A clear error message should also point out the annotation that caused the problem.

OrderMatters

If a class has many metacode associated with it, they can be called in an order that is not clear to the metaprogrammer [13] or they may be called in an order that prevents them from producing correct code or doing the intended checks.

An example of this issue is given using language Xtend [24]. Metacode are linked to *active annotations* and are processed in the order the annotations appear in each source file. Hence, if a source file uses annotatations `reqrLog`, `addLog`, `reqrLog`, `addLog`, in this textual order, the compiler will call the metacode associated to `reqrLog` first (two times) and then the metacode for `addLog` (two times).[3] This dependence of metacode call from textual order can introduce errors.

```
class Zero { String log; }
@addLog class First { }
@reqrLog class Second { String log; }
@addLog @reqrLog class Third { }
```

Assume that the active annotation `@reqrLog` requires that the class it is attached to has a field called `log`. In this example, `reqrLog` is attached to class `Second` that does have a `log` field. The active annotation should issue a compilation error for class `Third` because there is no such field. However, it does not because the `addLog` annotation adds to its attached class a `log` field.

Since Xtend calls the metacode based on the textual order of annotations, there will be no error because the metacode for `addLog` is called first and add a `log` field to both `First` and `Third`. Only after that the metacode for `reqrLog` is called and checks if `Second` and `Third` have such a field. Suppose now the above code is changed by the addition of annotation `reqrLog` to class `Zero`:

```
1 @reqrLog class Zero { String log; }
2 @addLog class First { }
3 @reqrLog class Second { String log; }
4 @addLog @reqrLog class Third { }
```

Now the metacode linked to `reqrLog` is run first for classes `Zero`, `Second`, and `Third`. Only after that the compiler calls the metacode for `addLog`. The metacode linked to `reqrLog` will issue an error in class `Second` because no `log` field will be found. We conclude that, in Xtend, the introduction or removal of an annotation in one place can affect the order the metacode will be called in another place.

There are two subproblems of OrderMatters. One is **DifferentViews**: different metacode may have different views of the base program. An example, cited by Palmer and Smith [13], considers a metacode `A` that adds to a class `X` a field for every class in the same source file. The field name is the class name in lower-case (`y` for `Y`). Initially, there is only class `X` in the file but a metacode `B` adds another class `Y`. If `A` is run before `B`, metacode `A` adds only field `x` to class `X`. If it is run after `B`, it adds fields `x` and `y`. If the semantics of metacode `A` is "adds a field to `X` for every class in the *final* source file, after all code addition made by metacode", then metacode `A` should be the last one to run. But many languages with support to metaprogramming cannot guarantee that.

In the above example, the metacode called afterward can view the changes caused by the metacode called previously. This is a problem because the calling order may not be clear and also because a change in the metacode textual order in a source file may change the calling order (as in the Xtend example). The developer does not expect that such subtle changes cause drastic code modifications.

Other subproblem of OrderMatters is **InvalidateChecks**. A metacode checks the program that is later changed by another metacode, invalidating the check. For example, metacode `CompanyStyle`

[3] See the discussion group, https://groups.google.com/forum/#!topic/xtend-lang/_RTAYBSTLMU

step 0 (original source code)

```
@addFieldName int age;
```

step 1 (metacode adds ageName)

```
@addFieldName int age;
@addFieldName int ageName;
```

step 2 (metacode adds ageNameName)

```
@addFieldName int age;
@addFieldName int ageName;
@addFieldName int ageNameName;
// till the end of time
```

Figure 3: Metacode produces metacode indefinitely. The dashed rectangle was produced by previous annotation

issues a compilation error if any class field uses underscore in its name. Metacode `AddColor`, run after `CompanyStyle`, introduces a field `color_name`. The check made by `CompanyStyle` is invalidated.

InfiniteMetaLoop Metacode may generate annotations added to the source code, which in turn may generate metacode and so on, creating an infinite loop. An example is shown in Figure 3. In a step 0, the compiler calls the metacode associated with annotation `addFieldName` attached to field `age`. The resulting code is in the rectangle of step 1 with the newly added code shown in the dashed rectangle, which is another field `ageName`. This new field is also attached to an annotation `addFieldName`. The compiler then calls the metacode associated with the annotation for `ageName` producing the code in the dashed rectangle of step 2, and so on. The problem here is that the compiler will never end its execution.

Nontermination

Metacode are called by the compiler. Therefore, if a metacode does not finish its computation, the compiler does not finish either. This is different from InfiniteMetaLoop because the cause of the nontermination is in the metacode itself, not in the code it generates.

Nondeterminism Metacode are not limited to interact with the source code or the compiler. They can interact with the file system, the network, and other running programs. That means metacode may be nondeterministic. As an example, the metacode associated with annotation `addFileHere` adds the content of a file to class `Company`.

```
@addFileHere("fieldsAndMethods.txt")
class Company { ... }
```

Different compilations may have different versions of class `Company` even if the source code of the whole program did not change. This may be what is intended. But it is nondeterministic anyway and, therefore, confusing to the developers.

The compilation order of the source files of an object-oriented program is given by its *class dependence graph* introduced in the issue WhoDependsOnWho. Any topological ordering of this graph

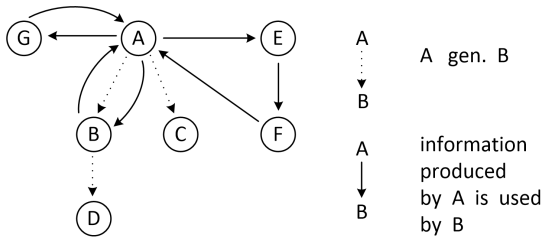


Figure 4: Two graphs: one showing metacode generation and the other showing which metacode uses information produced or changed by others

is a valid compilation order (there may more than one of them). Hence, the ordering is generally not unique and the compiler may choose different orders in different builds. This can be caused by the introduction of new files or changes in the last-modified date of source and binary files. The consequences are that the metacode may behave differently in two different builds. Let us show an example.

Class `Worker` has an annotation whose metacode adds methods to the class based on public methods of class `WorkContract`. Class `Company` has a similar annotation. Suppose that the classes are compiled, in two different builds, in two different topological orders:

```
build 1: Worker WorkContract Company
build 2: Company WorkContract Worker
```

If a metaobject adds methods to `WorkContract`, only `Company` will view these new methods in build 1 and only `Worker` will view them in build 2.

NoContracts

A metacode may demand specific features from the base code it is attached to and vice-versa [10]. For example, the metacode may demand the base class `T` declares a method for comparing two `T` objects. And the base code may demand that the metacode adds to the class a method `sort` (built with the comparison method). In this example, if either the base program or the metacode does not fulfill its part of the agreement, there will be a compilation error. But the compiler error message will not tell the developer that there was an agreement and which part has not fulfilled it. Errors may appear only in the final version of the source code which is a mix of base code with that added by metacode. To discover the errors, the developer has to examine the source file and scrutinize code generated by metacode, which exposes their implementation details.

A subproblem of NoContracts is **NoGeneratedCodeGuarantees**, which may happen because metacode can generate arbitrary code which may not match the metacode documentation. That means the generated code: (a) does not do what is intended; (b) does not compile; or (c) is malicious.

UnintendedCaptureId

This issue happens when metacode generates code that uses identifiers already in use in the environment where the code is inserted. Their semantics is equaled by accident. An example in Lisp, cited by Duba et al. [7], is the macro

```
(or e1 e2)
```

which should be expanded to

```
(let v e1 (if v v e2))
```

The expansion of

```
(or nil v)
```

results in

```
(let v nil (if v v v))
```

The last v in this expression was supposed to be different from the other occurrences of v (which are part of the macro definition). This problem is solved by renaming identifiers. When the metacode is a Lisp-like macro, this is called *macro hygiene*.

CircularDependency To explain this problem, we use two graphs whose vertices are metacode. A *code generation graph* (CGG) is a tree and there is a directed edge from A to B if metacode A generates code containing an embedded annotation linked to metacode B. Therefore, if there is an edge from A to B, the compiler runs metacode A that generates the annotation associated with B and, then, the compiler runs metacode B. In a *dependence graph* (DG), there is a directed edge from A to B if *base-program* information produced or changed by metacode A is used by metacode B. This information is *any property* of the base program such as the number of class fields, the superclass, or the presence or absence of a given method. An example of both the code generation and dependence graphs is shown in Figure 4. The CGG is obviously a tree and uses dashed edges. The edges of DC are continuous lines.

Problem CircularDependency happens when DG has a cycle. This is a problem because the compiler has to choose a metacode in a cycle to be the first to be run (of course, obeying first the CGG). Using the cycle A-E-F of Figure 4, suppose the compiler chooses metacode E to run first and then A and F. Metacode E generates code or does checks based on information that will be later changed when the compiler runs metacode A. The CircularDependency problem is an extended version of OrderMatters that is not solved by changes in the annotation order in the source code or an adequate choice of metacode execution order.

We will give an example of this problem with two annotations, rr and ss.

```
class P {
    @rr
    @ss
}
```

The metacode associated with annotation rr generates a field numFields initialized with the number of class fields (the original number of fields plus one because numFields is added too). The resulting code is

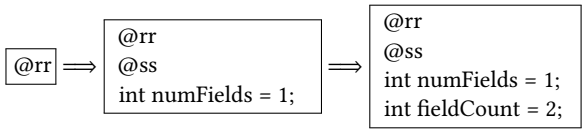
```
class P {
    @rr
    @ss
    int numFields = 1;
}
```

Now the compiler calls the metacode associated with annotation ss that generates the declaration of field fieldCount initialized with the number of fields of P, which is now 2. The resulting code is

```
class P {
    @rr
    @ss
    int numFields = 1;
    int fieldCount = 2;
}
```

This is wrong because the number of fields of P, in the final code, is 2 and, therefore, both fields should be initialized with 2. The problem here is that the metacode associated with rr and ss depend on the information, the number of fields, changed by both.

The circular dependence occurs even when the rr metacode generates the ss annotation:



This kind of dependence is illustrated by vertices A and B of Figure 4. Metacode A generates B and both depend on information generated or changed by each other.

WeakLinkAnnotMetacode

The language may not enforce a direct link between annotations in the program and the metacode. The developer does not know who metacode will be called for a given annotation in the source code. In Java [3], annotation processors (AP) are passed in the compiler command line. They can do checks in the program. Each annotation in the program is passed to each AP. If the processing method of one AP returns false, the annotation is passed to the next annotation processor. Therefore there is not a hard link between annotations and APs, making it difficult to associate semantics to an annotation.

AfterCompilation

Compile-time metaprogramming happens at compile-time but some later processing may be necessary. How to do this? This problem is presented with an example. A positive annotation is attached to a method and takes a single parameter which should be an int parameter to the method. The metacode assures that the real argument is greater or equal to zero.

```
class Person {
  @positive(age)
  public void setAge(int age) { ... }
  ...
}
```

The metacode may have two possible semantics. In semantics (A), the metacode inserts an if statement into setAge for assuring that age >= 0. This semantics does not demand any actions by the metacode after compilation because all it has to do is made at compile-time.

In semantics (B), annotation positive demands that only real arguments that are provable >= 0 at compile-time are legal to setAge. Suppose Person is compiled and put in a package L. Another source code U imports L, declares a variable of Person and sends a message setAge(n) to it. When the compiler finds this message passing, it has to call the metacode associated with positive that will analyze the AST of the callee method to discover if n

>= 0. We will say that "file U activates the metacode"; that is, the metacode is called in the environment of source file U because the semantics of its associated annotation demands that.

Assume that there is only one class per source file. We will define two terms: Src(Person) is the source file in which Person is and Act(Person) is a list of files that activate metacode associated with Person annotations. Usually this list has one file, which is Src(Person) itself. In the example and using semantics (B), there is at least one more file U in Act(Person). U is the source file that imported package L with class Person.

Based on the above example, we can now define the AfterCompilation issue as a list of related problems:

(1) if the compiler generates native code, how to add the metacode to binary files and how the linker deals with them?
(2) the metacode may need information of both Src(Person) and a file U ≠ Person. In the example, the metacode needs to check that the annotation parameter age is a formal parameter of method setAge, which is made at compile-time of Person. At a later time, after Person was compiled, the metacode needs to check that the real argument to the method is provable >= 0. These two activation times need two environments for metacode execution;
(3) how to interoperate two programming languages one of which supports annotations like positive that demands actions after the compilation time of the source file it is in?

ToolsDoNotKnowMeta

Tools that handle source code such as static analyzers and Integrate Development Environments (IDE) have to know about the metacode. Otherwise, they cannot work correctly. As an example, suppose a class declares a private field that is attached to an annotation that will create get and set methods for it. If the field is not used anywhere in the class, a static analyzer may point out that the field can be removed.

To know all of the possible consequences of the annotations used by a source code, a tool has to know everything about the metacode associated with the annotations. This is only possible if the metacode are run. But when they run, they can interact with compiler data structures such as the code AST. Since the set of metacode is not fixed, tools cannot discover metacode behavior. For that, they should be a superset of the compiler. This is the ToolsDoNotKnowMeta issue.

The original cause of this problem is that metacode change the language and, for each code with annotations, the developer is effectively using a different language that is unknown to the tools.

DeepUnexpectedChanges

Any sufficiently powerful metaprogramming system permits deep changes in the program by the metaprogram. For example, the metaprogram can modify the language semantics (as the meaning of message passings), remove methods and fields, alter the superclass of a class, remove implemented interfaces, remove method statements, add or remove method parameters, change method return type, and rename classes, fields, or methods. Profound changes make code unreadable. The developer has to know the meaning of every metacode in a source file before assuring very basic facts about that code. Therefore, the developer cannot be sure that a method that is in the source code does exist in the final code (the code produced after all changes made by the metacode). She or he cannot be sure a message passing obeys the expected semantics.

If incorrectly used, a powerful metaprogramming system can make the code very difficult to understand. However, it can also make the language support important features as is the case with CLOS [6], created to simulate several Lisp dialects.

3 POSSIBLE SOLUTIONS

This section discusses some possible solutions to the problems presented previously.

MessWithOthers. Figure 1 shows the interactions between annotations, metacode, and the compiler. To prevent MessWithOthers, the objects passed from the compiler to the metacode should only allow the latter to change the source code where the annotation is. Therefore, the compiler cannot pass to the metacode an object of its AST that gives access to AST objects of an external source file **and** allows the changing of it. But, to prevent this issue, the compiler should either never supply AST objects to metacode (too drastic!) or define alternative AST classes (another AST only for metacode). These other AST classes should not allow changes in external source files either because they supply a restricted view of external files or they check who is asking for the changes in the AST (if it is an external file, issue an error). This problem happens with OpenJava [21] [20] and languages supporting compiler plugins: Scala, Java, X10, Kotlin, TypeScript, and Rust. CLOS and OpenC++ supports *metaclasses* and a metaclass can only change the class that is its object. In BSJ [13], a security mechanism prevents non-local changes.

WhoDependsOnWho Any powerful metaprogramming system demands that code information flows from the compiler to the metacode and, therefore, has this problem. The only solution is to track every call to methods of the compiler objects made inside the metacode. If the call introduces a dependence from a source file R to S, the compiler-object method has to register that in the *class dependence graph* (CDG) kept by the compiler. As an example, the metacode associated with S calls a compiler-object method that gives the names of public methods of a class in R. Therefore, there should be an edge in the CDG from R to S. This problem happens with all languages cited in this paper except OpenC++. In this language, a class has no information on other classes.

KnowsFriendsSecrets The compiler objects passed to the metacode should never allow the leaking of private information from other source files. The solution to this problem is, therefore, similar to the solution to MessWithOthers. All languages that support a compiler plugin have this issue. OpenC++ does not because it limits the visibility to the class the metaclass is linked to.

Compiler-Interactions The object the compiler passes to the metacode should be carefully designed to prevent this issue. Therefore, the objects may be read-only or, if they are not, their methods should do checks to prevent any incorrect operation that can damage the compiler. The metacode could, instead of changing the compiler objects directly, *ask* the compiler to do the changes

through a Domain Specific Language (DSL). All languages with compiler plugins have this problem.

WhoDidWhat The compiler has to associate every metaobject with the changes it does in the program, as is done by the language Converge [22]. That can be done by several mechanisms: (a) the metacode ask the compiler to do code changes (they do not have the power of changing the code themselves);[4] and (b) the methods of the compiler objects track and register which metacode called every method that changed the code.

The compiler of the language Converge [22] tracks of who generates which code. Therefore, this issue is solved in this language. Converge generates bytecodes and each of them knows who produced it, which can be used even for runtime error messages. An AST object can be associated with more than one metacode that handled it. Therefore, the compiler error messages can point out all metacode that possibly caused an error.

OrderMatters The calling metacode order should be fixed and any changes to the code should only be visible in the next compilation phase. This guarantees that metacode called in the same compilation phase view the same source code and, therefore, their calling order is not important. All languages that allow handling of the AST, as the languages with compiler plugins, have this problem. After a metacode changes an AST object, all metacode view the changed object.

InfiniteMetaLoop This problem happens when the metaobject associated with annotation m_i generates code with annotation m_{i+1} for $i = 0, 1, ...$ The compiler can limit the maximum number of nestings to a finite number. This problem occurs with any language that, after compiling the code generated by metacode, processes any introduced annotation. The articles and websites about the languages cited in this text do not allow us to conclude which languages have this problem.

Nontermination The compiler calls the metacode in a separate process and kills it if it does not finish after some milliseconds. No language cited in this paper solves this problem. However, there are metaprogramming languages that do [16] as SafeGen [5], MorphJ [4], and Meta-traits [14].

Nondeterminism A total solution to this issue demands drastic measures: (a) the metacode is run in a container that prevents access to the external world (files, network); (b) all of the compiler objects passed as parameters to the metacode are read-only; and (c) OrderMatters is solved because it also introduces nondeterminism. To our knowledge, no annotation-based metaprogramming language solves this problem. Only very limited systems for metaprogramming like C++ templates [19] are deterministic.

NoContracts Ideally, there should be a *contract DSL* to specify the agreements between the metacode and the base code. Even so, the DSL code cannot specify everything. In particular, it can specify only part of the semantic requirements because checking the semantics of a (meta) code is incomputable. Therefore, the subproblem NoGeneratedCodeGuarantees can only be partially solved. If there is no contract DSL, metacode can check by themselves at least some of the demands they place on the base code. However, these demands would be more precisely described using a DSL code that is easily examined by the developer.

In its simplest form, the contract between the base code and the metacode could be just a Java-like interface, which is a set of method signature[5] declarations. The metacode demands that the class it is used with declares the methods of an interface `MM` and the base code demands that the metacode generates the methods of an interface `BB`. This is very similar to the requirements a generic class in Java and C♯ may place on its real arguments that are types:

```
class MyList<T extends ISorteable> { ... }
```

Here, `ISorteable` should be either implemented or inherited from the real argument type.

No language cited in this paper solves this issue. The metaprogramming language MTJ [14] is not annotation-based but it provides the best approach to this problem. It uses a feature called traits which are Java-like *interfaces*.[6] A *trait* has parameters and clauses `requires` and `provides` that specify the contract between the trait (which plays the role of an annotation) and the class that implements it (which plays the role of the source code the annotation is).

UnintendedCaptureId The compiler can supply to the metacode an object with a method for generating new variables that do not appear in the base code. However, the problem can continue if the metacode do not use this method. Another solution is to rename all of the free identifiers in the metacode generated code so they do not collide with the environment identifiers. If necessary, the metacode could ask for using some of the environment identifiers. No language cited in this paper automatically solves this issue.

CircularDependency This is an unsolvable problem in general because it is caused by intrinsic circularity in the system. However, each metacode may be aware of the code generated by other metacode and they can work together in search of an agreement. For example, in the first step all of the metacode that change a class may run and change the class. In a second step, each metacode may change the code it generated based on the modifications made by other metacode. That would continue in step 3, and so on till all of the metacode reach an agreement. Although the problem definition, by itself, does not allow a general solution, language BSJ partially solves this issue because it allows the developer to specify the execution order of the metacode. This forcibly eliminates circularity.

WeakLinkAnnotMetacode The language should specify how the annotations are linked to the metacode. For example, metacode are put in packages and, when the package is imported by a source file, the annotation names associated with the metacode can be used. Usually, this problem occurs in all languages supporting compiler plugins because, in these languages, the code that processes the annotations are put as options to the compiler.

AfterCompilation There is no general solution to this broad problem but some recommendations can decrease its dangerousness.

[4] This could be done with the DSL cited in Compiler-Interactions.

[5] A method signature specifies only the method name, parameter types, and return value type.

[6] A Java interface defines a set of methods with or without body and it does not inherit from the top-level class, `Object`.

First, the format of the binary code must be changed to accommodate the metacode information. Second, an annotation should be linked to two different metacode: one is run on the attached source code (where the annotation is) and the other in the activation source code. Each metacode may be a method of an object and, therefore, they share information using the object fields. To our knowledge, the compiled languages cited in this paper, X10 and Rust, avoid the problems by not allowing the metacode to act after a code is compiled.

ToolsDoNotKnowMeta Any static analyzer and IDE should be integrated with the compiler so they know which code the metacode generate and which checks they do. There is no other way of solving this problem. We are not aware of any tool that works correctly with a complete metaprogramming system.

DeepUnexpectedChanges Based on our own experience, deep changes in the program structure are rarely necessary. Therefore, it makes sense that metacode have limited power in changing the program. For example, metacode should not be able to delete any code (class, methods, fields, superclasses, etc), rename identifiers, and change the types of fields, variables, and method parameters. Metacode could do further code checks but not change the meaning of any construct of the base language. All languages with compiler plugins have this issue.

4 CONCLUSION

This paper listed some problems which are intrinsic to annotation-based compile-time metaprogramming and others which are flaws of current languages. It is difficult to put in a single category most of the problems. For example, MessWithOthers can be considered intrinsic because it is reasonable that the metaprogram changes the program. That includes the fact that a metacode associated with a file changes another one. But problem MessWithOthers can also be diminished if the language prohibits or limits the changes a metacode associated with a file do with another file. Hence, this issue can be, at least partially, solved by a language. There is no clear-cut distinction between the issues. For example, InfiniteMetaLoop causes Nontermination and it is expected that a metaprogramming system with the MessWithOthers issue has also KnowsFriendsSecrets.

Most of the issues of section 2 apply to some other flavors of metaprogramming, in particular, to compile-time metaprogramming. We focus on annotation-based compile-time metaprogramming (ABCTMP) because this approach demands a minimal amount of background knowledge and the examples can be easily understood. Programs can interact with other (meta) programs in countless ways using a myriad of language features. For that reason, we do not claim to have cited all of the problems with ABCTMP. To our knowledge, no list of metaprogramming issues has been published before.

REFERENCES

[1] Curtis Clifton and Gary T. Leavens. 2003. *Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy.* Technical Report 329. Computer Science Technical Reports - Iowa State University.

[2] Robertas Damaševičius and Vytautas Štuikys. 2008. Taxonomy of the Fundamental Concepts of Metaprogramming. In *Information Technology and Control.* Vol. 37. Kaunas University of Technology.

[3] Joe Darcy. 2006. Java Specification Request 269: Pluggable annotation processing API. http://jcp.org/en/jsr/detail?id=269

[4] Shan Shan Huang and Yannis Smaragdakis. 2011. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 6 (Feb. 2011), 44 pages. https://doi.org/10.1145/1890028.1890029

[5] Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2005. Statically Safe Program Generation with Safegen. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering* (Tallinn, Estonia) *(GPCE'05)*. Springer-Verlag, Berlin, Heidelberg, 309–326. https://doi.org/10.1007/11561347_21

[6] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. 1991. *The Art of Metaobject Protocol.* MIT Press, Cambridge, MA, USA.

[7] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. ACM, New York, NY, USA, 151–161. https://doi.org/10.1145/319838.319859

[8] Kotlin 2020. Compiler Plugins (Kotlin). https://kotlinlang.org/docs/reference/compiler-plugins.html

[9] Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (Oct. 2019), 39 pages. https://doi.org/10.1145/3354584

[10] Florian Lorenzen and Sebastian Erdweg. 2016. Sound Type-Dependent Syntactic Language Extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 204–216. https://doi.org/10.1145/2837614.2837644

[11] Nathaniel Nystrom and Vijay Saraswat. 2007. *An annotation and compiler plugin system for X10.* Technical Report. Technical Report RC24198, IBM TJ Watson Research Center.

[12] Oracle. 2023. Interface Plugin. https://docs.oracle.com/en/java/javase/11/docs/api/jdk.compiler/com/sun/source/util/Plugin.html

[13] Zachary Palmer and Scott F. Smith. 2011. Backstage Java: Making a Difference in Metaprogramming. *SIGPLAN Not.* 46, 10 (Oct. 2011), 939–958. https://doi.org/10.1145/2076021.2048137

[14] John Reppy and Aaron Turon. 2007. Metaprogramming with Traits. In *Proceedings of the 21st European Conference on Object-Oriented Programming* (Berlin, Germany) *(ECOOP'07)*. Springer-Verlag, Berlin, Heidelberg, 373–398.

[15] 2020. *Compiler Plugins (Rust).* https://doc.rust-lang.org/1.5.0/book/compiler-plugins.html

[16] Marco Servetto and Elena Zucca. 2013. A Meta-Circular Language for Active Libraries. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation* (Rome, Italy) *(PEPM '13)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/2426890.2426913

[17] Amanj Sherwany, Nosheen Zaza, and Nathaniel Nystrom. 2015. A Refactoring Library for Scala Compiler Extensions. In *A Refactoring Library for Scala Compiler Extensions (Lecture Notes in Computer Science, Vol. 9031)*, Björn Franke (Ed.). Springer, 31–48. https://doi.org/10.1007/978-3-662-46663-6.2

[18] Lex Spoon and Seth Tisue. 2020. Scala Compiler Plugins. https://docs.scala-lang.org/overviews/plugins/index.html

[19] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

[20] Michiaki Tatsubori. 1999. *An Extension Mechanism for the Java Language.* Master's thesis. University of Tsukuba, Japan.

[21] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. 2000. OpenJava: A Class-Based Macro System for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999.* Springer-Verlag, London, UK, UK, 117–133.

[22] Laurence Tratt. 2008. Domain Specific Language Implementation via Compile-time Meta-programming. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 31 (Oct. 2008), 40 pages. https://doi.org/10.1145/1391956.1391958

[23] typescript 2020. Using the Compiler API (TypeScript). https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API

[24] Xtend. 2020. Xtend — Modernized Java. https://www.eclipse.org/xtend/