Моделирование и анализ информационных систем. Т. 24, №6 (2017), с. 718–729
Modeling and Analysis of Information Systems. Vol. 24, No 6 (2017), pp. 718–729

DOI: 10.18255/1818-1015-2017-6-718-729

UDC 004.052.42, 004.4'6, 004.423.42, 004.432.2, 004.438 Eiffel, 519.681.2, 519.682.1

Towards Null Safety Benchmarks for Object Initialization

Kogtenkov A. V.

Received September 11, 2017

Abstract. Null pointer dereferencing remains one of the major issues in modern object-oriented languages. An obvious addition of keywords to distinguish between never null and possibly null references appears to be insufficient during object initialization when some fields declared as never null may be temporary null before the initialization completes. This work identifies the key reasons of the object initialization problem. It suggests scenarios and metrics to be used as the benchmarks to compare solutions of this problem. Finally, it demonstrates application of the benchmarks on the proposed solution for object initialization in Eiffel.

The article is published in the author's wording.

Keywords: null pointer dereferencing, null safety, void safety, object initialization, static analysis, null safety benchmarks

For citation: Kogtenkov A. V., "Towards Null Safety Benchmarks for Object Initialization", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 718–729.

On the author:
Alexander V. Kogtenkov, orcid.org/0000-0003-4873-8306, Doctor of Sciences ETH Zurich
Independent scientist, Podolsk, Russia, e-mail: kwaxer@mail.ru
address for correspondence: MAIS, 14 Sovetskaya str., Yaroslavl, 150003 Russia

1. Introduction

Null pointer dereferencing remains one of the major day-to-day issues in software industry. To construct a sound null-safe type system, most solutions of the problem for object-oriented languages add a notion of non-null and maybe-null types, usually expressed with additional type annotations. Such annotations would be sufficient to solve the null safety problem if objects could be created in an atomic operation, so that all fields marked as non-null were initialized with object references. Unfortunately, sequential initialization of the fields breaks the solution. Several proposals solving the object initialization issue [1, 2, 7, 9] suggest extending the type systems further to identify objects that are not completely initialized.

Instead of tweaking the type system, I proposed a static-analysis-based solution [10]. This solution relies on the validity rules checked during compilation to cover the cases left by the type system checks. Combined with removal of annotations for local variables [5], the solution is very effective in practice for avoiding null dereferencing problems, whilst

having low cost in compilation time: (a) it reduces the annotation overhead compared to previous solutions for the null-safe programming; (b) it correctly accepts or rejects code of the published examples relevant to this problem [1, 2, 7, 9]; (c) it permits new scenarios, impossible with type-system-based solutions.

But is the solution good enough? How does it compare to other approaches that solve the object initialization problem? In this work I review examples from the previous publications and identify the key reasons that cause the problem. Then I focus on the detailed criteria that can be used to compare different solutions.

The main contributions of this work are:

- identification of the roots of the object initialization problem;
- development of execution scenarios and metrics for benchmarks of different null safety solutions.

2. Motivating examples

Every publication on the problem of object initialization has few examples that authors use either to demonstrate issues with their approach, or to explain how the issues are resolved. I collected all the examples from the publications as well as found in class libraries and divided them into the following cases. I use the differences between some examples to describe common scenarios in section 5.

i. Polymorphic call from a constructor. Manuel Fähndrich and Rustan Leino [1] describe a call to a virtual method on `this` in a superclass constructor. Because subclass fields of the object are not initialized yet, accessing them in the polymorphic call causes `NullReferenceException`. Xin Qi and Andrew C. Myers [7] consider a similar example with a class `Point` and its subclass `CPoint` that adds a color attribute.

ii. Polymorphic callback from a constructor. Accesses to an uninitialized object can be done indirectly. If a superclass constructor passes a reference to the current object as an argument to create another object, this "remote" constructor can call-back on the object where not all fields are initialized yet.

iii. Modification of existing structures. Convenience of the ability to invoke regular procedures inside a constructor can be demonstrated with a mediator pattern [3]. It decouples objects so that they do not know about each other, but still can communicate using an intermediate object, *mediator*. If the communicating objects register themselves with the mediator in the code of their constructors, clients do not need to clutter their code with the calls to register every new communicating object after creating it.

iv. Safety violations. In addition to valid cases, authors usually mention examples that should trigger a compiler error. This aims at the original goal: a sound solution should catch potential null dereferencing at compile time.

v. Circular references. Manuel Fähndrich and Songtao Xia [2] review a linked list example with a sentinel. When a new list is constructed, a special sentinel node is created and it should reference the original list object.

vi. Self-referencing. This is a variant of circular references when an object references itself rather than another object.

3. Practical void safety

3.1. Why is object initialization problematic?

Null safety is complicated for object initialization. To understand why, I suggest to look at how program execution can lead to the null reference exception. Firstly, the object that causes the problem should not complete its initialization, i.e., some of its fields of non-null types should be null. Secondly, this object should be accessible — either directly or through some variables. Thirdly, the information that its initialization is incomplete should be lost. Otherwise, it would be easy to report the error at compile time. Finally, the reference retrieved from the uninitialized field of the object should be dereferenced to trigger the exception. To summarize, there are the following roots of the problem:

- *Non-atomic initialization* of an object leads to the possibility to have fields with null values even when their type is non-null.
- *Aliasing* allows for accessing the same object from an arbitrary point of the program, in particular, from the code that does not expect an incompletely initialized object.
- *Uncontrollable control flow*, interrupting the regular one, makes sequential reasoning about program execution useless.
- *Dereferencing* of an uninitialized field of the incompletely initialized object triggers the exception.

My review focuses solely on the null safe frameworks that use an existing object-oriented language as the basis. This aims at reusing legacy code if possible and preserving known coding techniques and patterns. In particular, I assume that constructors are allowed to execute arbitrary code, not just a sequence of assignments of supplied arguments to the fields. This implies that the current object can be used in the constructor and to escape from it as soon as such uses and escapes are guaranteed to avoid null dereferencing.

To achieve the safety, the solutions extending the type system with new types limit the operations on incompletely initialized objects. For Eiffel, I developed the *practical void safety* solution, briefly described in the next subsection. It specifies the conditions, when dereferencing may be unsafe, and forbids such dereferencing altogether.

3.2. Solution outline

From the point of view of the solution that avoids additional types annotations, all examples from section 2 can be divided into 2 major groups:

- Examples i to iv: – Can the code be reordered to initialize all fields before use?
- Examples v and vi: – Can compile-time rules ensure an object with recursive references to itself is not used as a completely initialized one?

The issue in the first group arises because the current object is passed before all fields of this object are set. The issue in the second group arises because not all fields can be set before passing a reference to the current object. Then, on the one hand, a call on an incompletely initialized object cannot assume all attributes are properly set. On the other hand, a qualified call does not allow seeing what operations on an incompletely initialized object are performed. The *practical void safety* solution disallows qualified calls when some objects are incompletely initialized:

Validity rule. *A constructor is null-safe if it satisfies all the following conditions:*

1. *All fields of the class are set at the end of the constructor.*
2. *Every field is set before it is used.*
3. *Any of the following is true at every execution point:*
   - 3.1 *All fields are properly set.*
   - 3.2 *The reference to the current object is unused before or at the current execution point.*
   - 3.3 *The expression at the execution point is neither of*
     - *a qualified call;*
     - *a creation expression that makes a qualified call.*

Here, the term "current object" means a special entity `this` (in C# and Java) or **Current** (in Eiffel). The term "qualified call" stands for the call with an explicit target (i.e., has the form `target.message`) whereas the term "unqualified call" is used for the call on the current object (i.e., has the form `message`). The rule applies not only to the constructors declared in the class, but also to the inherited ones. The latter should be rechecked in descendants even if the constructors are already checked in the classes where they are defined.

In these conditions and because types of objects that are initialized by the constructors are known, all unqualified calls from the constructors can be resolved at compile time. Therefore, the methods, involved in these calls, can be checked using the same validity rule. This allows for reusing initialization methods without any special annotations. Although the compile-time checks take advantage of the static method resolution, the generated code for constructors can still be shared among different classes and can rely on dynamic dispatch for unqualified calls.

4. Related work

Raw types ***(solve examples i and iv with*** 2+ ***annotations).*** Manuel Fähndrich and K. Rustan M. Leino [1] denote attached types with T^{-} and detachable types with T^{+} and propose to add raw types T^{raw-} to be used for partially initialized objects. If class C has an attribute of type T and some entity has type C^{raw-} then a qualified call to this attribute has type T^{+} regardless of original attachment status of that attribute. An assignment to an entity of a raw type accepts only a source expression of a non-raw non-null type to ensure that if an object becomes fully initialized, it cannot be

uninitialized. Also, by the end of every constructor, every non-null field should be assigned. Unfortunately, the rules for super-class constructors are not directly applicable to languages with multiple class inheritance, like Eiffel. Also, this approach does not support creation of circular references.

Masked types ***(solve examples i to vi with many annotations).*** Xin Qi and Andrew C. Myers [7] address the complete object life cycle. They instrument the type system with so called "masks" representing sets of fields that are not currently initialized. For example, the notation `Node\parent!\Node.sub[l.parent] -> *[this.parent]` for an argument `l` tells that it has a type `Node` and on entry requires that its field `parent` is not set and at the same time fields declared in subclasses of `Node` are not set unless `l.parent` is initialized. On exit the actual argument conforms to the type `Node\*[this.parent]` that indicates that the node object will be completely initialized as soon as its field `parent` is set. The notation is very powerful and goes far beyond null safety, but even with its complexity authors complain that it is not sufficient for real programs.

With masked types the results of a flow-sensitive type analysis are checked against provided specifications, while in *practical void safety* approach they are used to check the Validity rule conditions.

Free and committed types ***(solve examples i and iv to vi with*** 1+ ***annotations).*** Alexander J. Summers and Peter Müller [9] set the following design goals:

1. *Modularity:* the type system can check each class type separately.
2. *Soundness:* the type system is safe, i.e., null pointer exceptions are impossible at run-time.
3. *Expressiveness:* the type system handles common initialization patterns. In particular, it allows objects to escape from constructors and supports the initialization of cyclic structures.
4. *Simplicity:* the type system is simple and requires little annotation overhead.

The authors distinguish just two object states: under initialization and completely initialized. A newly allocated object has a so called "free" type. When an object is deeply initialized, i.e., all its fields are set to deeply initialized objects, it is said to have a "committed" type. The commitment point logically changes the type of an object from free to committed and is defined as the end of a constructor that takes only committed arguments. Possible aliasing between free and committed types is prevented by not having a subtyping relation between them. This differs from the convention for raw types [1].

The Validity rule is very close in spirit to the idea of free and committed types. But it relies on a flow-sensitive analysis and ceases free type status when all attributes are set. This allows the *practical void safety* to handle cyclic data structures without explicit annotations.

Other approaches ***(solve examples i to vi with*** 0 ***annotations, non-modular).*** Additional annotations are avoided by Bertrand Meyer in [6] using so called "targeted expressions" and creation-involved features. The analysis is somewhat similar to the abstract interpretation approach used by Fausto Spoto [8] and should be applied

to the system as a whole, thus sacrificing modularity. The advantage of the non-modular checks is in accepting larger code base as correct, i.e., in better expressiveness.

5. Benchmark criteria

The most important goal of the null safety design is soundness. This limits the possibilities to write arbitrary code that is still null safe. Indeed, the general problem of safe object initialization is undecidable. Therefore, some restrictions should be imposed on the code to make sure it can be checked in finite time.

As usual, soundness and expressiveness work against each other: the simpler the language rules, the less code can be written without violating them. If the rules are too strict, some scenarios found in real software can become extinct. E.g., the *raw types* [1] do not allow for creation of circular structures, and the *free and committed types* [9] rule out registration of objects in existing object structures inside constructors.

I use the design goals suggested by Alexander J. Summers and Peter Müller [9] as the base criteria to evaluate usability of null safety solutions. I give a detailed analysis of every aspect of the goals and review how it applies to distinguish characteristics of different approaches.

5.1. Soundness

Authors of all the solutions [1, 2, 7, 9] from section 4 claim them to be sound, so do I [10] for the *practical void safety*, based on the Validity rule. Unfortunately, not all aspects of a real programming environment are usually reflected in the formal model. In particular, none of the null safety formalizations reflects garbage collection that is an important channel to compromise safety guarantees.

The roots of the object initialization problem, mentioned in section 3, are mapped to the programming language constructs as follows:

- *Non-atomic initialization* corresponds to the order of initialization of the object fields intermingled with other computations. This work does not consider languages that support atomic (transactional) object initialization.

- Explicit *aliasing* becomes possible when an object is assigned to a field of an existing object, either passed to the constructor as an argument or directly reachable from the current context, or when the new object is thrown as an exception. Implicit *aliasing* happens when the class declares a finalizer that gets access to the object.

- *Uncontrollable control flow* can be caused by concurrent execution, preemptive execution (with exception and signal handlers), cooperative execution (coroutines).

- *Dereferencing* is done by a qualified call of the form `target.access` where `access` stands for a field or method name and `target` is a name of a reference corresponding to one of uninitialized field of the object.

Contrary to the formal models, programming languages, supporting some null safety mechanisms, do not always handle object initialization properly. A notable example is Kotlin [4] where, at the time of writing, null safety is unsound.

```
class X create make feature
    item: detachable A
    put (value: A) do item := value end
    make
        do
            (create {A}.initialize (Current)).
                data.do_nothing
        rescue
            if attached item as value then
                value.data.do_nothing
            end
        end
end
```

```
class A inherit
    EXCEPTIONS
create initialize feature
    data: A
    initialize (x: X)
        do
            x.put (Current)
            raise (Void)
            data := Current
        end
end
```

Fig 1. An example (in Eiffel) of a buggy scenario A-I(1a)

A. Escaping of uninitialized objects

If a program context does not expect an uninitialized object, there should be no channels that allow for the object to escape to this context. The channels depend on the programming language. The most common ones are discussed next.

A-I. Registration in an existing object. A program can register a new object in an existing one. When this is done before the new object is completely initialized, there is a problem: the incompletely initialized object can be accessed via the exiting object because of aliasing. Thus, such registration should be disallowed. The scenario can be further classified by

1) the *source* of the reference to the existing object that can be either (a) passed as an argument to the constructor or (b) retrieved from the current context (static data, once functions, singleton objects);

2) the *type* of the object in which the new one is registered: it can be (a) a user-defined one or (b) a built-in one (e.g., an array).

A registration of the new object in the existing one requires a qualified call. The condition 3.3 of the Validity rule disallows any qualified calls until all objects become completely initialized. Therefore, the *practical void safety* solution guarantees that the unsafe scenario with the explicit registration is impossible.

The example corresponding to this scenario is shown in fig. 1. It can be used in the suit of benchmarks for null safety solutions. The method *make* of the class X creates an object of the type A and passes a reference to itself (let's call it x) as an argument. The constructor *initialize* of the class A saves a reference to the current object in the object x. At this point the field *data* is null. Then a raised exception breaks the execution of the constructor in *initialize*. The rescue clause in *make* intercepts the exception, retrieves the incompletely initialized object and makes a call on the field *data* of a non-null type. This causes the null reference exception.

A-II. Reclamation of incompletely initialized objects. Finalizers are the methods called before object's memory is reused. The finalizers are registered for calling by the run-time right after object's memory is allocated and before the constructor is invoked. If the object initialization does not complete (due to an uncaught exception in the constructor), the finalizer is invoked on the incompletely initialized object. Unless a programmer keeps track of object initialization, there is no way to figure out what state the object is in. Therefore, the current object in a finalizer should be treated like at the beginning of a constructor.

A-III. Out-of-order object transfer. Most programming languages allow for transferring references to objects bypassing regular control flow. The most familiar mechanism is exceptions. If a new exception object referencing an incompletely initialized one is thrown, the reference to the incompletely initialized object becomes accessible in the code that relies on the type system rules and does not expect uninitialized fields.

There is no special construct to raise an exception in Eiffel. A qualified call to a library method is used to do it. According to the condition 3 of the Validity rule, all objects should be completely initialized at this point, therefore, soundness is preserved.

B. Non-sequential control flow

Approaches solving the object initialization problem with whole-system analysis make a safe approximation about all possible execution traces. Consequently, these approaches are non-modular.

The modular approaches are limited by the local analysis only and should assume worst-case scenarios for operations on incompletely initialized objects. This is achieved by pretending that sequential execution can break anywhere. In other words, the solutions do not exclude unexpected interruption. Therefore, here I just list possible scenarios that can be used to build the benchmark tests:

B-I. *Exceptions.*

B-II. *Concurrency.*

B-III. *Cooperative execution.*

C. Dereferencing

C-I. Unqualified calls. Access to uninitialized fields should either be prohibited or result in a potentially null value. The condition 2 of the Validity rule sticks to the first variant.

C-II. Qualified calls. Before all objects are completely initialized, language rules should either disallow qualified access to fields or make sure the retrieved values are not considered as safe for use. In particular, these values may be null or (recursively) have null values in non-null fields of referenced objects. The *practical void safety* solution takes the first route and disables qualified calls until all objects are completely initialized.

5.2. Expressiveness

Any example demonstrating a soundness issue from section 5.1 can be turned into a legitimate one. To this end, one of the conditions that cause the problem should be

```
data := Current
x.put (Current)
raise (Void)
    (a)
```

```
raise (Void)
data := Current
    (b)
```

Fig 2. Possible fixes of the method *initialize* from fig. 1

made false. E.g., setting the field *data* before passing a reference to the current object as in fig. 2a or avoiding leaking this reference altogether as in fig. 2b, both make the example from fig. 1 legitimate.

D. Calls on incompletely initialized objects

Calls on incompletely initialized objects require special precaution, but they allow for more code reuse.

D-I. Unqualified calls.

1) *Method call.* One of the key uses of unqualified calls is initialization itself. The methods that set object fields can be called from different constructors to avoid code duplication.

2) *Field access.* For the same reason, access to the fields that might be unset can be useful outside of object initialization. The code would check if the field value is null and proceed as needed.

D-II. Qualified calls. In general, qualified calls on incompletely initialized objects are unsafe. Therefore, additional restrictions should be imposed on the code of the called methods as well as on the code of the callers. They both cannot rely on the regular type system rules.

1) *Method call.* The called methods cannot assume that fields of non-null types are not null.

2) *Field access.* Safe access to object fields can be guaranteed only if they have a basic type without nested references.

E. Circular references.

Object structures with circular references appear to be a common design pattern. They can be classified by the structure kind:

1) *Self-referencing.* A new object references itself after creation.

2) *Mutual references.* (a) Two objects or (b) n objects make a circular structure after creation, where n is not necessary fixed at compile time.

3) *Complex structures.* A combination of cases 1) and 2).

F. Regular use of a completely initialized object from the constructor

F-I. Callback. When an object is completely initialized, it can be passed from the constructor as an argument to create another object that makes a callback later. This pattern is used in some portable GUI libraries.

F-II. Escaping of initialized objects. The scenarios repeat the scenarios A:

1) *Registration in an existing object.*

2) *Out-of-order object transfer.*

But now the object is completely initialized and does not cause a problem.

G. Genericity

Solutions can differ by the ability to use a formal generic type as a creation type. For the *practical void safety* solution, special convention should be introduced to indicate whether the constructor of the actual generic parameter satisfies the Validity rule requirements.

5.3. Modularity

H. Scope. The solution is modular if it is sufficient to analyze (recursively) ancestors and suppliers of the class to be checked.

The Validity rule depends on the properties of the constructors from the other classes. Because these constructors are known at compile time, the checks do not depend on the classes that are not directly reachable from the one being checked. Therefore, a library can be checked as a standalone entity without the need to recheck it after inclusion in some other project.

I. Incrementality. The metric tells if changes to previously checked code require a partial recheck rather than a complete one.

With the *practical void safety*, fast recompilation is supported if information about reachable constructors and whether they perform qualified calls is recorded for every class.

5.4. Simplicity

This group of metrics tells how accessible is the solution.

J. Number of additional annotations. This is the number of different type marks that need to be added besides the marks “non-null” and “maybe-null”.

K. Ease. This metric describes whether few new simple rules are added to the language to make object initialization null safe.

L. Performance. The metrics describes the resource consumption to support the additional checks. This includes:

1) **space complexity** – how much additional memory is required;

2) **time complexity** – how much time the new language checks take.

M. Availability. This metric tells if there is a tool (compiler/framework) that supports the rules.

6. Preliminary results

The table below summarizes results of selected benchmarks for different solutions (a plus sign indicates a positive result of the benchmark, a minus sign indicates a negative one):

Solution	**ABC**	*D-I(1)*	*D-I(2)*	*D-II*	**E**	**F**	**H**	**J**	**K**	**M**
Raw types	+	+	+	+	−	−	+	2+	+	+
Masked types	+	+	−	+	+	+	+	many	−	−
Free/committed types	+	+	+	+	+	−	+	1+	+	+
Targeted expressions	+	+	−	+	+	+	−	0	±	−
Practical void safety	+	+	−	−	+	+	+	0	+	+

All the solutions are sound (if the scenario *A-II* is not taken into account) and support simple use cases. With modularity and annotation overhead in mind, the best two solutions are *free & committed types* and *practical void safety*. Unqualified access to uninitialized attributes (*D-I(2)*) can be easily supported in the latter solution. But inability to support registration of a newly created object from the constructor in an existing object (**F**) with the former solution cannot be fixed easily. Therefore, *practical void safety* seems to be a better choice to solve the object initialization problem.

7. Conclusion and future work

In this work, I identify the roots of the object initialization problem in object-oriented languages and propose a list of benchmarks to compare different null safety solutions. The benchmarks demonstrate very good results for the *practical void safety* compared to other solutions.

The work reveals the following areas of future development:

- formalization of null safety models taking into account finalizers and possible interruption of execution due to asynchronous signals;
- improvement of the *practical void safety* solution by supporting access to uninitialized attributes and by supporting qualified calls in the context with incompletely initialized objects;
- creation of a test suit with executable examples to compare different implementation of null safe programming languages.

References

[1] Fähndrich Manuel, Leino K. Rustan M., “Declaring and Checking Non-null Types in an Object-oriented Language”, *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA ’03, ACM, New York, NY, USA, 2003, 302–312, http://doi.acm.org/10.1145/949305.949332.

[2] Fähndrich Manuel, Xia Songtao, “Establishing Object Invariants with Delayed Types”, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA ’07, ACM, New York, NY, USA, 2007, 337–350, http://doi.acm.org/10.1145/1297027.1297052.

[3] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] JetBrains, *Kotlin Language Specification*, https://jetbrains.github.io/kotlin-spec/kotlin-spec.pdf, visited on 2017-01-31.

[5] Kogtenkov A. V., "Mechanically Proved Practical Local Null Safety", *Proceedings of the Institute for System Programming of the RAS*, **28**:5 (2016), 27–54, DOI: 10.15514/ISPRAS-2016-28(5)-2.

[6] Bertrand Meyer, *Targeted expressions: safe object creation with void safety*, 2017, http://se.ethz.ch/ meyer/publications/online/targeted.pdf, visited on 2017-05-08.

[7] Qi Xin, Myers Andrew C., "Masked Types for Sound Object Initialization", *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, ACM, New York, NY, USA, 2009, 53–65, http://doi.acm.org/10.1145/1480881.1480890.

[8] Spoto Fausto, "Precise null-pointer analysis", *Software & Systems Modeling*, **10**:2 (2011), 219–252, http://dx.doi.org/10.1007/s10270-009-0132-5.

[9] Summers Alexander J., Müller Peter, "Freedom Before Commitment: A Lightweight Type System for Object Initialisation", *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, ACM, New York, NY, USA, 2011, 1013–1032, http://doi.acm.org/10.1145/2048066.2048142.

[10] Kogtenkov Alexander, "Practical Void Safety", *Verified Software. Theories, Tools, and Experiments*, 2017, http://dx.doi.org/10.1007/978-3-319-72308-2_9.

Когтенков А. В., "К критериям оценки безопасности нулевых ссылок при инициализации объекта", *Моделирование и анализ информационных систем*, **24**:6 (2017), 718–729.

DOI: 10.18255/1818-1015-2017-6-718-729

Аннотация. Разыменование нулевого указателя остаётся одной из основных проблем в современных объектно-ориентированных языках. Очевидное добавление ключевых слов, чтобы различать всегда ненулевые и возможно нулевые ссылки, оказывается недостаточным во время инициализации объекта, когда некоторые поля, объявленные всегда ненулевыми, могут временно быть нулевыми до окончания инициализации. Данная работа устанавливает ключевые причины проблемы инициализации объектов. Она предлагает сценарии и метрики в качестве эталонных тестов для сравнения решений этой проблемы. Наконец, она демонстрирует применение этих тестов к предложенному решению инициализации объектов в Eiffel. Статья публикуется в авторской редакции.

Ключевые слова: разыменование нулевого указателя, безопасность нулевых ссылок, безопасность пустых ссылок, инициализация объектов, статический анализ, эталонные тесты безопасности нулевых ссылок

Об авторе:
Когтенков Александр Валентинович, orcid.org/0000-0003-4873-8306, кандидат наук (Doctor of Sciences ETH Zurich), Независимый ученый, г. Подольск, Россия, e-mail: kwaxer@mail.ru
адрес для корреспонденции: редакция журнала МАИС, ул. Советская, 14, г. Ярославль, 150003 Россия