Safe Navigation in OCL

Edward D. Willink[1]

Willink Transformations Ltd, Reading, England,
ed@willink.me.uk

Abstract. The null object has been useful and troublesome ever since it was introduced. The problems have mitigated by references in C++, annotations in Java or safe navigation in Groovy, Python and Xbase. Introduction of a safe navigation operator to OCL has some rather unpleasant consequences. We examine these consequences and identify further OCL refinements that are needed to make safe navigation useable.

Keywords: OCL, safe navigation, multiplicity, non-null, null-free

1 Introduction

Tony Hoare apologized in 2009[3] for inventing the null reference in 1965. This 'billion dollar mistake' has been causing difficulties ever since. However NIL had an earlier existence in LISP and I'm sure many of us would have made the same mistake.

The problem arises because the null object has many, but not all, of the behaviors of an object and any attempt to use one of the missing behaviors leads to a program failure. Perhaps the most obvious missing behavior is the navigation expression *anObject.name* which accesses the *name* property of *anObject*. Whenever *anObject* can be null, accessing its *name* property will cause the program to fail.

A reliable program must avoid all navigation failures and so must prove that the source object of every navigation expression is never null. This is often too formidable an undertaking. We are therefore blessed with many programs that fail due to *NullPointerException* when an unanticipated control path is followed.

Language enhancements such as references[2] in C++ allow the non-nullness of objects to be declared as part of the source code. Once these are exploited by good programmers, compile-time analysis can identify a tractably small number of residual navigation hazards that need to be addressed.

A similar capability is available for Java using annotations such as *@NonNull*[5], however problems of legacy compatibility for Java's large unannotated libraries makes it very hard to achieve comprehensive detection of null navigation hazards.

An alternative approach pursued by languages such as Groovy[4], Python[7] and Xbase[8]. A safe navigation operator makes the nulls less dangerous so that *anObject?.name* avoids the failure if *anObject* is null. The failure is replaced by a

null result which may solve the problem, or may just move the problem sideways since the program must now be able to handle a null *name*.

In this paper we consider how OCL can combine the static rigor of C++-like references with the dynamic convenience of a safe navigation operator. In Section 2 we introduce the safe navigation operators to OCL and identify that their impact may actually be detrimental. We progressively remedy this in Section 3 by introducing non-null object declarations, null-free collection declarations, null-safe libraries, null-safe models and considering the need for a deep non-null analysis. Finally we briefly consider related work in Section 4 and conclude in Section 5.

2 Safe Navigation Operators

OCL 2.4 has no protection against the navigation of null objects; any such navigation yields an invalid value, which is OCL's way of accommodating a program failure that other languages handle as an exception. OCL provides powerful collection operators enabling compact expressions such as

```
aPerson.father.name.toUpper()
```

This obviously fails if `aPerson` is null. It also fails whenever `father` is null as may be inevitable in a finite model. A further failure is possible if `name` is null as may happen for an incomplete model.

2.1 Safe Object Navigation Operator

We can easily introduce a `?.` operator to OCL by defining `x?.y` as a short-form for

```
if x <> null then x.y else null endif
```

We can rewrite `aPerson.father.name.toUpper()` for safety as

```
aPerson?.father?.name?.toUpper()
```

This ensures that the result is either the expected value or null; no invalid failure.

2.2 Safe Collection Navigation Operator

Collections operation are a very important part of OCL and any collection navigation such as

```
aPerson.children->collect(name)
```

will fail if any element of the children collection is null.

We can easily introduce a `?->` operator to OCL by defining `x?->y` as a short-form for

```
x->excluding(null)->y
```

We can rewrite the problematic collection navigations for safety as:

```
aPerson?.children?->collect(name)
```

This ensures that any null children are ignored and so do not cause an invalid failure.

2.3 Safe Implicit-Collect Navigation Operator

The previous example is the long form of explicit collect and so could be written more compactly as:

```
aPerson.children.name
```

The long form of the `?.` operator in `x?.y` is therefore

```
x->excluding(null)->collect(y)
```

We can rewrite for safety as

```
aPerson?.children?.name
```

This again ensures that null children are ignored.

2.4 Assessment

OCL 2.4 already has distinct object and collection navigation operators, with implicit-collect and implicit-as-set short-forms. These are sufficient to confuse new or less astute OCL programmers, who may just make a random choice and hope for a tool to correct the choice. Adding a further two operators can only add to the confusion. We must therefore look closely at how tooling can exploit the rigor of OCL to ensure that safe navigation eliminates one of OCL's limitations. FIXME ASBH I'm not sure which OCL limitation you refer there

2.5 Safe Navigation Validation

The safe navigation operators should assist in eliminating errors and the following tentative Well Formedness Rules can identify an appropriate choice.

Error: Safe Navigation Required. If the navigation source could be null, a safe navigation operator should be used to avoid a run-time hazard.

Warning: Safe Navigation not Required. If the navigation source cannot be null, a safe navigation operator is unnecessary and may incur run-time overheads.

The critical test is *could be null / cannot be null*. How do we determine this for OCL?

Some expressions such as constants `42` or `Set{true}` are inherently not null. These can contribute to a program analysis so that a compound expression such as `if ... then Set{42} else Set{} endif` is also non-null even though we may not know anything about the if-condition. Unfortunately, OCL permits any object to be null and so all accesses to objects can be null. In practice this means that most OCL expressions cannot be usefully analyzed and the validation WFRs will just force users to write `?.` everywhere just to silence the irritating errors.

3 Non-null declarations

We have seen how the safe navigation operator fails when non-null objects cannot be usefully identified. We will therefore examine how to identify such objects.

3.1 Non-null Object declarations

We could consider introducing non-null declarations analogous to C++ reference declarations. We could even re-use the `&` character. But we don't need to, since UML already provides a solution and a syntax. When declaring a TypedElement, a multiplicity may qualify the type:

```
mandatoryName : String[1]
optionalName : String[?]
```

`[?]` indicates that a String value is optional; a null value is permitted.

`[1]` indicates that a String value is required; a null value is prohibited.

OCL can exploit this information coming from UML models and may extend the syntax of iterators, let-variables and tuple parts to support similar declarations in OCL expressions. However, since OCL has always permitted nulls, we must treat `[?]` as the default for the new OCL declarations even though `[1]` is the default for UML declarations.

With respect to use of the `*` unlimited natural literal expression as TypedElement multiplicity, we will add some comments in the next section where null-free collection declarations are introduced.

3.2 Null-free Collection declarations

The ability to declare non-null variables and properties provides some utility for safe navigation validation, but we soon hit another problem. Collection operations are perhaps the most important part of OCL, and any collection may contain none, some or many null elements. Consequently whenever we operate on collection elements we hit the could-be-null hazard.

Null objects can often be useful. However collections containing null are rarely useful. The could-be-null collection elements hazards are therefore doubly annoying, there is an irritating hazard diagnosis and the diagnosis complains about typical usage. FIXME ASBH I don't understand what you want to say in this last sentece.

In order to eliminate the hazard diagnosis, we must be able to declare that a collection is null-free; i.e. that it contains no null elements. This could be treated as a third boolean qualifier extending the existing ordered and unique qualifiers. We could therefore introduce the new names, NullFreeBag, NullFreeCollection, NullFreeOrderedSet, NullFreeSequence and NullFreeSet but this is beginning to incur combinatorial costs.

A different aspect of UML provides an opportunity for extension. UML supports bounded collections, but OCL does not, even though OCL aspires to UML alignment. The alignment deficiency can be remedied by following a collection declaration by an optional a UML multiplicity bound. Thus `Set(String)` is a short-form for `Set(String)[0..*]` allowing UML bounded collections and OCL nested collection to support e.g. `Sequence(Sequence(Integer)[3])[3]` as the declaration of a 3*3 Integer matrix.

However, this UML collection multiplicity tells nothing about if elements *cannot be null*. We require an extension of the UML collection multiplicity, to also declare an element multiplicity. Syntactically we can re-use the 'where' operator (FIXME ASBH 'where' operator name looks like wrong. I'm unsure about how to name it without using "vertical bar symbol" or similar, though.)to allow `[x|y]` to be read as 'a collection of multiplicity x where each element has multiplicity y'. We can now prohibit null elements and null rows in our example Integer matrix by specifying `Sequence(Sequence(Integer)[3|1])[3|1]`.

Finally, we are getting somewhere. A collection operation on a null-free collection obviously has a non-null iterator and so the known non-null elements can propagate throughout complex OCL expressions. Provided we use accurate non-null and null-free declarations in our models, well written OCL does not need any change. Less well written OCL has its null hazards diagnosed. (FIXME ASBH It's not clear to me what is well written OCL and why it doesn't need any change.)

With respect to the usage of the `*` unlimited natural literal as a possible element multiplicity, in principle, a declaration as the following one:

```
multipleNames : String[*]
```

would be unnecessary in OCL since we already have the OCL collections to denote that a property comprises multiple values. Therefore the notation, above would be equivalent (shorthand) to declare the following one:

```
multipleNames : Collection(String)[*|?]
```

in which we are specifying that *multipleNames* can have between 0 and an unlimited number of Strings, and due to the OCL defaults, all of them *could be null*.

Therefore, it turns out that the only possible literals that we can use for element multiplicities are `[1]` and `[?]`, specifically to declare the possible non-null-ness (null-free-ness in case of being used as collection element multiplicity) or null-ness of that element.

3.3 Null-safe libraries

The OCL library provides a variety of useful operations and iterations, but they have never been modeled. We have only indicative unchecked manual declarations, which lack the precision we need for null-safe analysis.

Consider the declaration

```
String::toBoolean() : Boolean
```

Using the default legacy interpretation that anything can be null, this should be interpreted as

```
String::toBoolean() : Boolean[?]
```

We have an additional postcondition:

```
post: result = (self = 'true')
```

Intuitively this assures a true/false result. But we must always consider null and invalid carefully. If `self` is null, the comparison using `OclAny::=` returns false, and if `self` is invalid the result is invalid. We are therefore able to provide a stronger backward compatible library declaration that guarantees a non-null result.

```
String::toBoolean() : Boolean[1]
```

We can pursue similar reasoning to provide `[?]` and `[1]` throughout the standard library.

We hit problems where the non-null-ness/null-free-ness of a result is dependent on the non-null-ness/null-free-ness of one or more inputs.

Consider the notional declaration for Set::including.

```
Set(T1)[c1|e1]::including(T2)(x2 : T2[e2]) : Set(T3)[c3|e3]
```

The requirement for T3 to be the highest common type of T1 and T2 can be satisfied directly by changing the declaration to

```
Set(T)[c1|e1]::including(x2 : T[e2]) : Set(T)[c3|e3]
```

If e1 and e2 are Boolean-values with true for `[1]` (is not null) and false for `[?]` (may be null), e3 is just `e1 and e2`. If our library modeling support Boolean-valued OCL equations to determine the result multiplicity, we can model this and avoid the need for an implementation to transliterate specification words into code.

Preliminary discussions (FIXME ASBH Preliminary discussions ? where ? reference ?) indicated limited enthusiasm for accurate modeling of collection bounds in OCL, so we could just take the view that OCL does not support bounded collections enthusiastically; c3 is always `[0..*]`. However if we need to support equations for the element multiplicity, we can easily support equations for the pessimistic collection multiplicity too.

```
c3 = c1->first() .. c1->last()+1
```

3.4 Null-safe models

Once the standard library has accurate null-safe modeling we are just left with the problem of user models.

For object declarations, there seems little choice but to make this part of the user's modeling discipline; object declarations must accurately permit or prohibit the use of null.

For collection declarations the default may-be-null legacy behavior is mostly wrong and for some users it may be universally wrong. We would like to provide a universal change to the default so that all collections are null-free unless explicitly declared to be null-full. In UML, we can achieve this by applying an `OCL::Collections` stereotype to a Package or Class. The `nullFree` Boolean stereotype property provides a setting that is 'inherited' by all collection-valued properties within the Package or Class.

UML has no support for declaring collection elements to be non-null, so we need a further `OCL::Collection::nullFree` stereotype property to define whether an individual TypedElement has a null-free collection or not.

For disciplined modelers, the sole cost of migrating to null-safe OCL will be to apply an OCL::Collections stereotype to each of their Packages.

3.5 Deep non-null analysis

Accurate non-null declarations enable WFRs to diagnose null navigation hazards ensuring that safe navigation is used when necessary. However simple WFRs provide pessimistic analysis.

For instance, the `anObject.name` navigation in the following example is safe since it is guarded by `anObject <> null`

```
let anObject : NamedElement[?] = ....
in anObject <> null implies anObject.name <> null
```

However a simple WFR using just `anObject : NamedElement[?]` diagnoses a lack of safety because the `anObject` let-variable may be null. A potentially NP complete program flow analysis is needed to eliminate all possible false unsafe diagnostics. A simpler pragmatic program flow analysis can eliminate the common cases of an if/implies/and non-null guard.

4 Related Work

The origin and long history of null problems has been alluded to in the introduction as has the mitigation for C++ and Java.

The safe navigation operator is not new since at least Groovy, Python and Xbase provide it.

The possibility of safe navigation in OCL is new, or rather the pair of `?.` and `?->` operators were new when we suggested it in the Aachen workshop[1]. The utility of the `[?]` and `[1]` non-null multiplicities was also mentioned at the

Aachen workshop. The null-free declarations, stereotypes and the interaction between safe navigation and non-null multiplicities have not been presented before, although they are available in the Mars release of Eclipse OCL [6].

5 Conclusions

We find that naive introduction of safe navigation to OCL risks just doubling the number of arbitrary navigation operator choices for an unskilled OCL user. These problems are soluble with tool support provided we can also solve the problem of declaring non-null objects and null-free collections.

We take inspiration from UML multiplicity declarations to provide the necessary declarations. Due to the fact that by default OCL collections can contain null values, we exploit stereotypes so that the impact for well-designed models may be as little as

- one stereotype per Package to specify that all of its collections are null-free
- an accurate `[?]` or `[1]` multiplicity to encode the design intent of each non-collection Property (FIXME ASBH is this well contextualized ? i.e. it doesn't seem ot be related to the Stereotypes you are talking about, is it ?)

References

1. Brucker, A., Chiorean, D., Clark, T., Demuth, B., Gogolla, M., Plotnikov, D., Rumpe, B., Willink, E., Wolff, B.: Report on the Aachen OCL Meeting . Comm. Pure Appl. Math. 33, 609–633 (1980)
2. Ellis,M., Stroutstrup, B.: The Annotated C++ Reference Manual. (1990)
3. Hoare, T.: Null References: The Billion Dollar Mistake. QCon London (2009)
4. JSR 241: The Groovy Programming Language. (2004)
5. Using null annotations: http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-using_null_annotations.htm
6. Eclipse OCL: https://www.eclipse.org/modeling/mdt/downloads/?project=ocl
7. Python Software Foundation: The Python Language Reference. 2.7.10 (2015)
8. Xbase: https://www.eclipse.org/Xtext/documentation/305_xbase.html#xbase-expressions