

# A Validity Analysis to Reify 2-valued Boolean Constraints - Extended Abstract

Edward D. Willink<sup>1</sup>

<sup>1</sup>Willink Transformations Ltd, Reading, England

## Abstract

As an executable specification language, OCL enables metamodel constraints that cannot be sensibly expressed graphically to be resolved textually. However many users have expressed disquiet that although a constraint is obviously either satisfied or not, the OCL formulation is not 2-valued. We argue that this disquiet is the consequence of a misunderstanding emanating from the failure of the OCL specification to address crashing. We introduce an analysis that identifies potentially invalid computations and so guarantees that Constraints are 2-valued and that OCL-based Model Transformations do not malfunction.

## Keywords

Program Validation, Model Transformation, OCL, Crash

## 1. Introduction

The full, too-long, version of this paper may be found at [1].

As a Model-Oriented Pseudo-Code, OCL has been successfully used by many researchers to elaborate models within their Domain-Specific Utopia. However OCL is also an executable specification language and in this role significant problems arise in respect of what happens when evaluation goes wrong. We will use the unambiguous but emotive term ‘crash’.

Many languages have a mechanism whereby a crash results in a thrown exception that bypasses all further execution until explicitly caught by a suitable handler. OCL has a potentially equivalent mechanism whereby an `invalid` value is passed from the crash to an `oclIsInvalid` handler. However, it is only equivalent when all the intervening execution that processes the `invalid` value is *strict*. Any `invalid` input leads to an `invalid` output.

Programmers are familiar with the short-circuit idiom whereby evaluation of the first argument of an `and` / `or` operation can make evaluation of the second argument redundant and so avoid a crash. The OCL specification defines `and` / `or` operations to be commutative and so does not support short-circuit operations. In order to emulate short-circuit behaviour, strictness is abandoned and the two-valued operators are elaborated to be four-valued.

OCL therefore has the strange execution behaviour that a crash can occur while evaluating one term of a Boolean operator only for it to be ‘uncrashed’ by the other term. This is at best surprising and inefficient. In practice it presents significant challenges for tooling that must provide a capability to unwind a crash. These challenges grow for OCL-based applications such as model transformations in which unwinding may require an ability to `unmatch`.


---

OCL 2021: 20th International Workshop on OCL and Textual Modeling, June, 2021, Bergen, Norway

✉ ed at willink.me.uk (E. D. Willink)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

An examination of the crashes that can occur in OCL identifies two varieties.

Catastrophic crashes such as `PowerFail`, `MemoryFail`, `NetworkFailure`, `IOFail`, `StackOverflow` can occur at almost any time and there is very little that an ordinary OCL program can do about them. It is therefore very desirable that all such crashes should always crash so that the application that invokes the failed OCL can provide a helpful diagnosis.

Other crashes such as `DivideByZero`, `NullObjectNavigation`, `ArrayIndexOutOfBounds` occur predictably and are the result of deficient programming. Bad programming is of course undesirable and so we should look to exploit the rigor of OCL to prevent such crashes ever occurring. We introduce a Validity Analysis so that all undesirable crash hazards are flagged as errors.

Once undesirable crashes are eliminated, the need for desirable crashes to always crash requires that all execution is strict. The Boolean operators must exhibit conventional short-circuit behaviour at run-time. This introduces an incompatibility when the crashing argument is evaluated before, or concurrently with, the guarding argument. Our Validity Analysis must therefore identify one of the arguments as crash-proof and commute the arguments at compile time so that the crash-proof argument is evaluated first avoiding the need to unwind a crash from a guarded crashable term. In the event that both arguments are crashable, deterministic execution can be ensured by computing the arguments to a pair of let-variables so that both crash hazards are resolved sequentially before the Boolean evaluation.

## 2. Validity Analysis

The Validity Analysis identifies all terms in an OCL expression that may crash, i.e. *MaybeInvalid*. Since this analysis is performed at edit- or compile-time, the actual values of many terms are unknown and so a symbolic evaluation is necessary to propagate knowledge such as  $\{Is, Maybe, Not\}$   $\{Empty, Invalid, Null, Zero\}$  through the OCL expression AST.

Taking a simple example with two crash hazards.

```
self.count > 0
```

A catastrophic or desirable crash can occur if the target model is hosted by a database allowing the navigation from `self` to `count` to experience a network failure. OCL can do nothing about this so we want the crash to happen and do not need to do anything to prevent or diagnose it.

A programming error or undesirable crash can occur if the multiplicity of the `count` property is `[?]` allowing a `null` value. This value will crash when `self.count > 0` is evaluated. Our Validity Analysis must diagnose this. We can guard the undesirable crash.

```
self.count <> null implies self.count > 0
```

Except that this assumes that OCL has conventional short-circuit semantics which it doesn't. For OCL 2, the crash should happen and then be unwound.

Once we revise the Boolean operators to be short-circuit, we need our Validity Analysis to have sufficient understanding of the program flow to recognise that the `self.count > 0` is only executed when its expression ancestors permit it. In our example this occurs when the first argument of the `implies` is true.

For this simple idiomatic example, the guard is obvious, but if we try to understand why it is obvious we find that we are performing a reverse evaluation from the one required true result of `self.count <> null` to establish the characteristics of the two inputs. This reverse evaluation is not monotonic and so does not scale to non-trivial examples.

Our Validity Analysis pursues an alternative approach that needs only forward evaluation.

A naive analysis of the example identifies that `self.count` *MaybeNull*, and consequently that `self.count > 0` *MaybeInvalid*. We need to demonstrate that the *MaybeInvalid* is *NotInvalid* to suppress the naive diagnosis,

The *MaybeInvalid* is the consequence of a precondition failure for the `>` operation and so we can hypothesize that the strictness precondition requiring non-null arguments is violated. i.e. we hypothesize that the `>` operation execution can occur with `self.count` is null. The ‘can occur’ aspect of the hypothesis imposes restrictions on all ‘if’ and ‘short-circuit’ ancestors. In our example, execution of the second term of `implies` mandates that the first term is true giving an additional constraint (`self.count <> null`) = `true`. Re-evaluation of all terms affected by the hypothesized value encounters a contradiction between the false-valued evaluation of `self.count <> null` for the hypothesized null value and the true-valued evaluation imposed by the executable control path. The contradiction refines the symbolic value of `self.count` when accessed within `self.count > 0`. *MaybeNull* changes to *NotNull* and so allows the symbolic re-evaluation to refine `self.count > 0` from *MaybeInvalid* to *NotInvalid*. The spurious crash hazard diagnosis is eliminated.

The example demonstrates the usage of symbolic *Null* and *Invalid* knowledge. This together with *Zero* and *Empty* is just about working in the Eclipse OCL prototype. Further work is needed to expand the aggregate coverage to handle aggregate *Size* and *Content* knowledge so that for instance `seq->includes(x)->first()` is hazard-free.

The Validity Analysis can never be powerful enough to understand arbitrarily complicated control flow and so users may need to help by making guards more explicit or by adding additional invariants. As a last resort, an additional `src.oclAssert(body)` may be required to allow a contextual constraint body to apply in the context of the `src`.

### 3. Conclusion

OCL’s crash handling can be refined to ensure that desirable crashes always crash and that undesirable crashes never happen.

The challenge is to ensure that the benefits outweigh the pains. Guaranteed freedom from undesirable crashes is a very significant benefit. Adding clarifying invariants may be painful.

Once OCL programs are free of undesirable crashes, OCL Boolean evaluations will appear to be 2-valued just as they appear to be in other languages.

### References

- [1] Willink, E.: A Validity Analysis to Reify 2-valued Boolean Constraints.  
<http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2021Validity/OCLValidity.pdf>