A Validity Analysis to Reify 2-valued Boolean Constraints

Edward D. Willink[1]

[1] 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

OCL [10] evolved from Syntropy to satisfy the need to elaborate UML [12] diagrams with constraint details that could not sensibly be expressed graphically. Within the context of a UML model, OCL specifies what happens within a domain-specific Utopia where nothing bad happens, not even when the user models real problems.

OCL is not just a model-oriented pseudo-code. OCL is a specification language that is also executable. The OCL specification makes some concessions to realizability by prohibiting infinite collections and tolerating indeterminacy for operations such as `Set::asOrderedSet()`. However there is very little consideration of what happens when things go wrong; the single solution of an `invalid` value is used for all problems.

In this paper we give detailed consideration to how OCL goes wrong and refine the specification so that when OCL goes wrong, it does so usefully and predictably.

We will use the emotive term crash for going wrong, since all programmers understand what a crash is. It avoids any confusion with terms such as invalid/exception/error/failure that may be associated with particular solution approaches.

While reviewing the many ways in which an OCL evaluation can crash, we identify the need for more than an ill-considered one-size-fits-all solution. In particular, we identify that there is nothing that can, or should be done, for some crashes, and so it is very desirable for these to be reliably propagated for resolution by the user. In contrast other crashes are the undesirable consequence of inadequate programming. We introduce a validity analysis to

OCL 2021: 20th International Workshop on OCL and Textual Modeling, June, 2021, Bergen, Norway
ed at willink.me.uk (E. D. Willink)

identify the inadequacies and so guarantee that OCL will always crash desirably and never crash undesirably.

Once we introduce rigor to OCL's crashing, we are forced to confront the conflicts between recursion, commutativity and short-circuits for the Boolean `and` and `or` operations. We refine their specification to remove the conflicts. Commutativity is only supported where the validity analysis proves that the usage *isCommutable*. We observe that commutativity is not actually necessary if pairwise idempotence is replaced by multi-term redundancy pruning.

In Section 2 we review the ways in which OCL can crash so that in Section 3 we outline what we need to achieve and in what respects the OCL specification needs to be tweaked. Section 4 presents a running example to show how even the simplest of invariants may be unsafe. In Section 5 we introduce the analysis and symbolic evaluation that diagnoses all crash hazards and in Section 6 we identify opportunities for better practice that exploits the validity analysis. In Section 7 we describe how far the implementation work has progressed. Finally in Section 8 we review the related work and conclude in Section 9.

2. Crashes

Programmers in most languages are resigned to the need to debug their programs to fix bugs and to handle exceptions where problems are unavoidable. OCL has no exception capability. Rather than throwing an exception as an out-of-band 'return', OCL returns the `invalid` singleton value as an in-band result. In principle, these two mechanisms are equivalent, particularly if a practical OCL implementation supports a richer `invalid` that includes details of the problem while continuing to behave as a singleton.

Many OCL programmers are unhappy that the in-band return of `invalid` means that an OCL Constraint is not 2-valued despite being a self-evident arbiter of whether some condition is satisfied or not. This unhappiness is actually a misunderstanding [8] since any non-trivial constraint expressed in almost any language has three possible outcomes; satisfied, not-satisfied and crash. The misunderstanding arises because, when the crash uses an exception mechanism, the crash outcome bypasses the invocation code, which perceives only two outcomes. In contrast, the OCL programmer must ensure that the invocation propagates the `invalid` back to the invocation's caller. The misunderstanding is therefore an ergonomic issue whereby the API provided by the OCL evaluator fails to meet the expectations of the programmer, and fails to alert the programmer to the simple solution of converting an OCL `invalid` result into an exception to propagate the crash.

It would clearly be better if programs do not crash, but before we look at reasons for OCL to crash, we will look at mechanisms that avoid some crashes.

2.1. Crash Avoidance

2.1.1. Well Formedness Rules

OCL expression terms such as *PropertyCallExp* navigate a model in accordance with its metamodel, which provides a strong type system with which the OCL expressions must comply.

Compliance is defined by the Well Formedness Rules that can check that for instance the type of the *ownedSource* of a *PropertyCallExp* conforms to the *owningClass* of its *referredProperty*.

An OCL validator should check all the WFRs, preferably at edit time, but at least before execution, since execution is likely to fail miserably if a WFR is violated. We may therefore assume that no crash occurs as a consequence of a WFR violation.

2.1.2. Guards

Where a programmer is aware that a crash may occur, the programmer may guard against it. A substantive guard may use an `if then else endif` clause to provide alternative functionality, or a more localized guard may use a logical operator.

```
(var != null) || var.doSomething() // C or Java
```

The Java above uses the short-circuit `||` operator to prevent a *NullPointerException* when `doSomething()` is invoked when `var` is `null`.

As we shall see, the equivalent OCL operator is not short-circuit. Rather than preventing a crash, it can allow a crash to happen and then require the crash to be uncrashed.

2.2. Catastrophic / Desirable Crashes

Problems such as Power Failure, Stack Overflow or Memory Allocation Failure can occur at almost any time and there is nothing that a normal OCL program can do about them.

Problems such File Access, Network or Database failure may occur when a *NavigationCallExp* requires an additional model element to be available. Again there is very little that a normal OCL program can do about them.

These problems are pretty catastrophic. We categorize the consequent crashes as desirable since the most sensible response is to diagnose the problem as helpfully as possible in the hope that the user may understand and resolve the issue.

We will revisit Stack Overflow in Section 2.5.

2.3. Careless / Undesirable Crashes

OCL supports a `null` value to reify the content of slots with optional multiplicities and an `invalid` value to reify the consequence of an evaluation failure. These values are not suitable for computation and so OCL defines a strict semantics whereby usage in *IteratorExp*, *OperationCallExp* or *NavigationCallExp* is a failure that results in an `invalid` value result. The loose wording in the specification could be formalized by preconditions for the evaluation counterpart of the expression.

```
context NavigationCallExpEval
pre ValidSource: not source.oclIsInvalid();
pre NonNullSource: not source.oclIsUndefined();
```

Failure of a precondition is a consequence of careless programming. We therefore categorize it as an undesirable crash. The programmer needs assistance to ensure that such crashes never occur.

The strict execution semantics of the OCL Abstract Syntax Tree elements provides a simple crash-and-stay-crashed behavior. The OCL Standard Library defines

- regular operations with preconditions
- irregular not-strict logical operations
- special operations that may use `OclVoid` or `OclInvalid` types

The additional preconditions provide further opportunities for careless programming.

2.3.1. Divide-by-Zero

The problem of divide-by-zero exists in many languages, but is relatively rare in practice and often easily avoided. OCL is little used for arithmetic, so the problem hardly exists in practical OCL, but it would nonetheless be nice to avoid the crash.

2.3.2. Index-out-of-bounds

The `OrderedSet` and `Sequence` collection types support indexing in much the same way as `Array` and `List` in other languages. A crash occurs when an unsuitable index access is used. This problem occurs more often than might be expected, since many users accidentally use the 0-based index typical of an execution language, rather than the 1-based index of a specification.

2.3.3. Missing Content

The collection types support reverse indexing using the `indexOf` operation, or the `any` iteration, and crash when the indexing misses. The crash from `indexOf` is excessive since a `null` or negative return could signal the query-miss less forcefully. It is unreasonable to expect every use of `indexOf` to be guarded by an `includes`.

2.3.4. Bad String Content

Operations such as `String::toReal()` support the lexical conversion of a string to a more interesting type. They crash if the source string is incompatible with the conversion. This crash is again excessive since a `null` could signal the conversion failure. It is impractical to expect the source string to always be lexically valid and completely pointless to require the user to write their own parser to be used in a guard.

2.4. Uncrashing

Once a crash has occurred, the programmer may take some action to handle it.

2.4.1. Catching

In many languages a crash is propagated by throwing an exception and subsequently catching it. In OCL, the crash is propagated as the `invalid` value and may be 'caught' by the `OcAlny::oclIsInvalid()` operation.

```
let result : OclAny = functionThatMayCrash() in
if result.oclIsInvalid() then fixupCrash() else result endif
```

Accommodating `OclAny::oclIsInvalid()` is inconvenient when realizing OCL by translation to a conventional language, since the conventional exception passing must be diligently trapped and converted to an `invalid`values wherever `oclIsInvalid()` might be invoked.

Ideally the usage of `oclIsInvalid()` would be limited to not-invalid preconditions and Operating System level OCL that really wants to catch a catastrophic failure to produce a friendly diagnostic or to perhaps retry on another computer.

2.4.2. Reverting

The avoidance of crashes by short-circuit operators in conventional languages was described in Section 2.1.2. Unfortunately the equivalent logical operators in OCL were specified to be commutative. The incompatibility between commutativity and short-circuiting was 'resolved' by making the logical operators not-strict to allow them to handle `null` or `invalid`. The commutativity is mathematically elegant but the consequent 4-valued {`true`, `false`, `null`, `invalid`} Boolean is unpopular with users and has bad implementation consequences.

The conventional short-circuit suppresses the unwanted evaluation of the second term.

```
(var != null) || var.doSomething() // C or Java
```

The hazardous second term is not evaluated; no crash occurs.

The OCL short-circuit is

```
(var <> null) or var.doSomething()
```

Since the operator is commutative, an implementation has a free choice of the evaluation order, and may even use different processors to evaluate the two arguments concurrently. For less obvious OCL expressions, it may be unclear to user or tooling what the best evaluation order is. An implementation cannot in general avoid evaluating the 'wrong' argument first. Subsequent evaluation of the 'right' argument may provide the guard value and so require the implementation to discard the 'erroneous' crash.

Even if the implementation foregoes the concurrency opportunity and evaluates first argument first, the commutativity allows a programmer to accidentally specify the guard second, so the implementation must still support the uncrash. Of course no sensible programmer will program the guard term second so the implementation is just being forced to implement something that should never happen.

Except that it does. During development, it is not uncommon for the system or at least the OCL exposition to be defective. A user who has set a breakpoint in code associated with a crash may find the debugger stopping at the crash and be confused when that crash fails to propagate as expected. The problem is that the crash during the first term evaluation may be inhibited by a malfunction in the second input evaluation. The overall execution may be pedantically correct, but at best CPU time has been wasted by crashing. More likely the developer spends significant time understanding the strange behavior possibly concluding, with some justification, that OCL execution is unreliable.

Unfortunately the commutative not-strict logical operators break the simple crash-and-stay-crashed behavior.

2.5. Stack Overflow Revisited

In Section 2.2 we lumped a Stack Overflow together with other catastrophic crashes that the OCL programmer can do nothing about. Certainly, once such a crash occurs, it is desirable to diagnose it, but the origin of the crash is frequently due to an uncontrolled recursion and so down to bad programming.

Detecting the soundness of an arbitrary recursion requires a solution to the Halting Problem which in general does not exist. We can however diagnose wherever an unsafe recursion hazard exists.

As a minimum we can identify recursive call sites. In practice the recursion will be guarded to ensure that it terminates. For relatively simple step and repeat recursions we may be able to analyze the step to see whether it iterates towards a limit and so remove the pessimistic unsafe characterization.

If the recursion terminates with the help of a short-circuit *and* or *or* operation we have a further conflict with commutativity; it is essential that the termination guard is evaluated before the next recursion starts.

2.6. Model Transformation

Many model transformation tools provide a disciplined framework to create or mutate an output model using immutable OCL queries on the input model. OCL crashes pose a difficult problem.

Some transformation languages such as QVTo [11] provide a relatively conventional exception mechanism allowing the users to handle OCL's `invalid` as an exception.

For declarative transformations, functionality is modularized by rules within which OCL specifies the matches and conversions. Execution is determined by the successful rule matches, so potentially an OCL crash just loses a rule match and the user is disappointed that some conversion did not happen. This is dishonest. Any crash is a transformation execution failure and any subsequent result is a suspect compromise. A declarative model transformation must crash enthusiastically.

When interpreting or generating code for a model transformation, the implementation must faithfully realize all possible OCL failures so that no crash is hidden. This requires considerable effort to support a behavior whose result is going to be thrown away. Much better to alert the programmer to all the undesirable crashes so that only desirable crashes remain allowing for a much simpler execution in which any crash is a fatal crash.

3. Goal

We have motivated our goal for normal OCL

- catastrophic/desirable crashes always crash
- careless/undesirable crashes never occur

This is fully in accord with OCL's strict behavior, provided preconditions are always satisfied. We need a validity analysis that can guarantee this proviso.

Unfortunately the OCL specification bundles a *short-circuit-and* behavior with a *commutative-and* behavior as a composite *and* operation. The conflicts between these behaviors and the rest of the specification are 'resolved' by undermining strictness.

However, the not-strict commutative specification of the logical operators conflicts with both our goals.

- A catastrophic crash can be guarded and so not crash.
- A careless crash may occur before it is guarded and uncrashed.

We can satisfy our goals by revising the logical operators to be sequentially strict. A strict evaluation of the first argument can ensure the crash happens. The first argument can then short-circuit the unwanted second argument evaluation guaranteeing that unwanted crashes do not occur.

This approach re-instates strictness to support a qualified form of commutativity. An alternative approach to repairing the specification that retains unqualified commutativity must compromise short-circuiting. This would be much simpler but more damaging to compatibility and expressiveness. It would eliminate everything potentially associated with short-circuiting.

- No recursion
- No null values for Boolean operations
- No invalid values for Boolean operations

3.1. Revised and operation

Taking the and operation as an example, we are changing the result of Table A.2 of the OCL specification from:

Use Case	Input 1	Input 2	Output
2-valued	true	true	true
	true	false	false
	false	true	false
	false	false	false
Normal Short-Circuit	false	ϵ	false
	false	$\perp$	false
Commutated Short-Circuit	ϵ	false	false
	$\perp$	false	false
Crash	true	ϵ	ϵ
	true	$\perp$	$\perp$
	ϵ	true	ϵ
	$\perp$	true	$\perp$
	ϵ	ϵ	ϵ
	$\perp$	ϵ	$\perp$
	ϵ	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$

to

Use Case	Input 1	Input 2	Revised Output
2-valued	true	true	true
	true	false	false
Normal Short-Circuit	false		false
Crash	true	ϵ	$\perp$
	true	$\perp$	$\perp$
	ϵ		$\perp$
	$\perp$		$\perp$

Normal short-circuit and 2-valued functionalities have unchanged results but now explicitly avoid the redundant second argument computation. This guarantees that no desirable crash is computed and then discarded.

The subtle change that all crashes return $\perp$ (`invalid`) rather than sometimes ϵ (`null`) is a reversion from the idempotence introduced in OCL 2.4 back to the simpler OCL 2.3.

3.2. Qualified Commutativity

The significant change is that the Commutated Short Circuit functionality now crashes on the first argument without giving the second argument a chance to discard a crash. Our validity analysis must identify this usage and commute the arguments at compile-time to ensure that a crash-proof argument is evaluated first.

An unavoidable but diagnosable incompatibility arises only when both arguments may crash. The diagnosis will force the programmer to sequence the potential crashes by caching at least one of them in a let-variable.

3.3. Commutativity Utility

It seems helpful to review to what extent unqualified commutativity is actually necessary.

A simple traditional example of the utility of commutativity comes from mental arithmetic where `3*15` is easier to calculate than `15*3`, since people are more familiar with their 3 rather than 15 times tables. More generally the combination of associativity, distributivity and commutativity may allow a calculation to be refactored to minimize inaccuracies when subtracting large nearly-equal numbers.

For Boolean arithmetic, ease of calculation is irrelevant, but a refactoring exploiting idempotence is essential to allow elimination of redundant terms. However commutativity is only necessary because the pairwise definition of idempotence needs to shuffle terms to enable `A & B & C & B & A` to be simplified to any permutation such `B & C & A`. If we replace 2-term idempotence by a multi-term idempotence in which a later repeated term is redundant, the example simplification has the unique `A & B & C` result that supports the optimization while ensuring that a guarding term precedes a guarded term.

Unqualified commutativity is therefore unnecessary. The validity analysis outlined in this paper enables the *isCommutable* precondition to be evaluated to determine when both arguments are crash-proof and so permit qualified commutativity. This will be most useful at compile-time where an expensive analysis can be tolerated to impose a smart strategy on the evaluation. Terms could be ordered according to some heuristic such as

- most-likely-to-guard-first
- most-expensive-to-compute-last
- most-likely-to-contribute-to-a-common-subexpression-first

The resulting OCL Abstract Syntax can capture the compile-time optimization. The run-time can blindly follow the defined order, since it is unlikely that the run-time can afford the overhead of determining a more optimum order and fairly unlikely that the run-time has extra profiling information to allow a better decision than at compile-time.

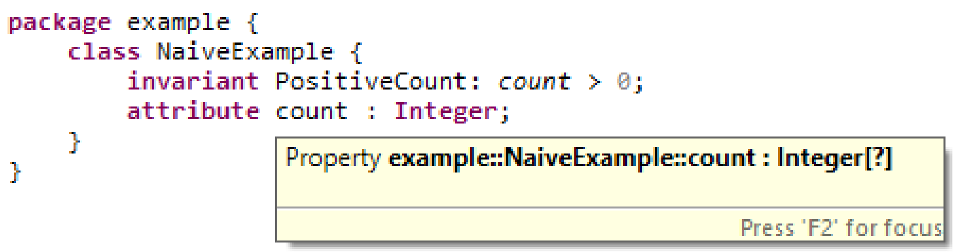
If concurrent evaluation of commutative terms is required, it will be necessary to augment the OCL *OperationCallExp* with a *may-be-concurrent* flag to capture the result of the compile-time analysis that guarantees that the terms are crash-proof and so commutable or parallelizable.

4. Running Example

Our running example considers the very simple class constraint shown using OCLinEcore [15] in Fig 1.

The `NaiveExample` class contains an *Integer* attribute named `count` and an invariant to require a positive value.

It would seem self evident that the invariant is 2-valued corresponding to satisfied/not-satisfied, but it is not. There are two crash hazards.

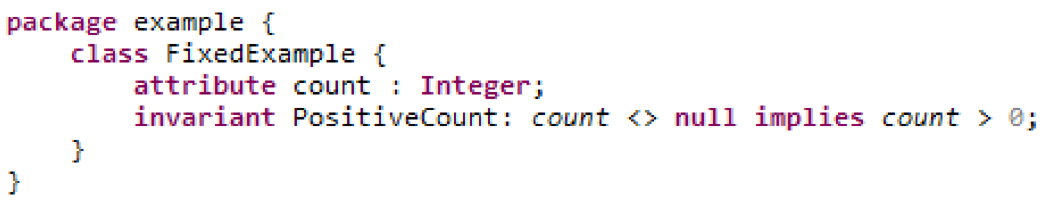


```
package example {
    class NaiveExample {
        invariant PositiveCount: count > 0;
        attribute count : Integer;
    }
}
```

Property **example::NaiveExample::count : Integer[?]**

Press 'F2' for focus

Figure 1: Naive Example



```
package example {
    class FixedExample {
        attribute count : Integer;
        invariant PositiveCount: count <> null implies count > 0;
    }
}
```

Figure 2: Fixed Example

4.1. Hazards

If the host model is served by a cloud network or database, there is a possibility that the *PropertyCallExp* access to `count` may fail with some form of network error. This error is treated as `invalid` by OCL and consequently the evaluation of the constraint yields an `invalid` result. As noted above, this rather pedantic but catastrophic concern is sensibly resolved by a strict any-crash-always-crashes philosophy.

Fig 1 also shows the OCLinEcore editor's hover text to reveal the underlying *Property* declaration. It has a fully qualified name `example::NaiveExample::count`, primitive type `Integer` and multiplicity `[?]`. The optional multiplicity allows the value of `count` to be `null`. In Ecore [7], where the emphasis is on simple default construction of Java objects, the default multiplicity lower bound for all objects is `0`. Consequently this is the OCLinEcore default and so a widespread practice. In contrast, for UML, the lower bound multiplicity default is unity so that a `null` is only permitted after an explicit user action. For either representation, a valid *Property* may specify that `null` is an acceptable value. The `null` value violates the strict precondition of the comparison operation. It crashes the invariant and disappoints the user hoping for a 2-valued outcome.

We require our tooling to support elimination of this not-2-valued hazard by diagnosing that the comparison operator requires non-null/non-invalid inputs but that an actual input *MayBeNull*.

4.2. Fixes

The user may easily fix the problem by correcting the optional `[?]` multiplicity to the non-optional `[1]`. Alternatively, if a `null` value is a required aspect of the design, the user may correct the invariant as shown in Fig 2.

The `implies` operation guards the comparison preventing the crash, but naively the tooling

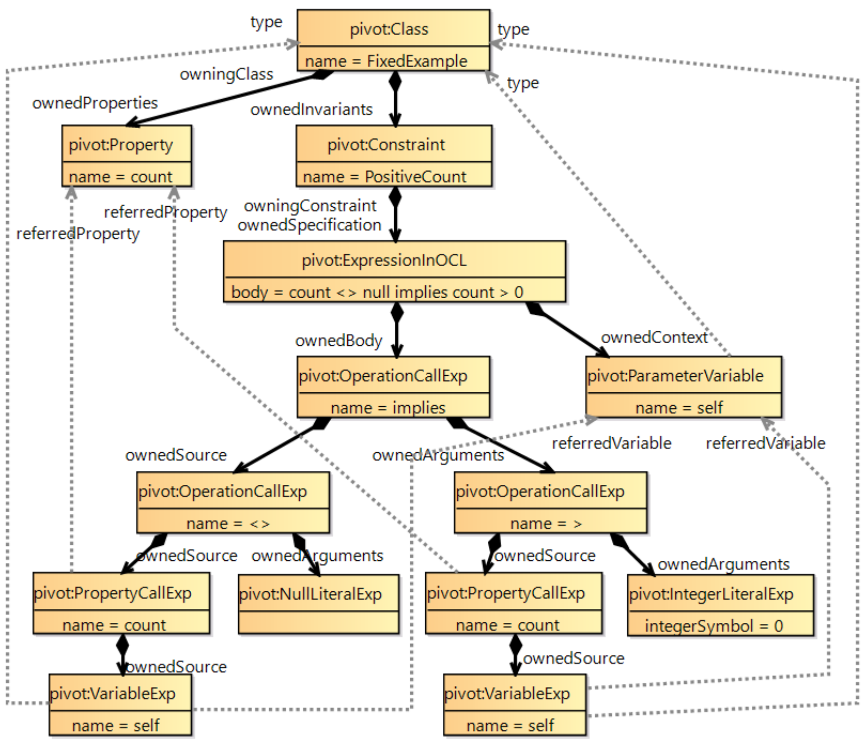


Figure 3: Fixed Example Abstract Syntax Tree

will continue to diagnose the hazard unless the tooling understands the program control flow consequences of the sequentially-strict `implies` operation.

5. Program Analysis

Our running example shows that even simple OCL code can have a problem that can be fixed. We now introduce an analysis to alert the user to the need for fixes, and to confirm that sufficient fixes have been applied to guarantee that no undesirable crashes occur, and that all desirable crashes always crash. We first review the conventional run-time evaluation of our example OCL expression.

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

5.1. Simple Evaluation

The OCL specification defines the Abstract Syntax of OCL expressions. Fig 3 shows the Pivot-based Eclipse OCL AST of the fixed example invariant using a UML Object Diagram-like exposition of the XML serialization. (Solid lines and diamonds for compositions, dashed lines for references, XML namespace:element-type box titles.)

The top of the diagram shows the *Class* named `FixedExample` with `count` *Property* and `PositiveCount` *Constraint* metamodel elements. The *Constraint* is realized by an *ExpressionIn-*

OCL with a `self` *ParameterVariable* and the *ownedBody* OCL expression tree with an `implies` *OperationCallExp* at its root[1].

The *ownedSource* sub-tree comprises another *OperationCallExp* for the <> operation with further sub-trees comprising a *PropertyCallExp* to evaluate the `count` property upon the result of the *VariableExp* that accesses the `self` *ParameterVariable*. The second sub-tree comprises a *NullLiteralExp* that evaluates to the `null` value.

The *ownedArguments* sub-tree of the `implies` comprises a very similar subtree to again evaluate `self.count` but to use a > operation to compare against the 0 *IntegerLiteralExp*.

At run-time this constraint may be used to confirm the well-formedness of each element of a user model. Each instance of `FixedExample` is bound in turn to the `self` *ParameterVariable* and then the *ownedBody* is evaluated by bottom up tree traversal with each descendant returning a result to its ancestor.

Execution therefore starts at the bottom left as a *VariableExp* accesses the `self` value and passes its value as the *ownedSource* for its parent *PropertyCallExp* that accesses the `count` slot and passes it as the *ownedSource* of the <>. The depth first traversal continues by providing `null` as the *ownedArguments* input of the <> from the *NullLiteralExp*. With both inputs computed, the <> can pass its result to as *ownedSource* for the `implies`, which once a similar traversal has computed its *ownedArguments* input can return the overall result to the *ExpressionInOCL*.

5.2. Precondition Evaluation

In OCL, operations such as > are strict requiring all inputs to be evaluated and to be non-invalid before execution. Whether operations also require non-null inputs is determined by the `[?]` or `[1]` multiplicity of each operation parameter. The specification wording can be formalized by preconditions.

```
operation Integer::>(arg : Integer[1]) : Boolean {
    precondition: not self.oclIsInvalid();
    precondition: not arg.oclIsInvalid();
    precondition: not self.oclIsUndefined();
    precondition: not arg.oclIsUndefined();
    precondition: self.oclIsKindOf(Integer);
    precondition: arg.oclIsKindOf(Integer);
}
```

A full evaluation should validate these and other preconditions by evaluating them. Eclipse OCL [13] never executes preconditions. USE [16] can do so when requested.

The conventional use of the OCL well formedness rules supports a static analysis that ensures that all input values are type compatible. Here we are concerned with a more extensive static analysis to ensure that all preconditions are satisfied. The static analysis occurs at edit/compile-time making evaluation at run-time redundant.

[1]The *referredOperation* links to the Standard Library model are omitted

5.3. Symbolic Evaluation

At edit/compile-time we have no actual instances, rather we need to prove that for all possible instances the result will be either 2-valued or a desirable crash. With our revision to strict semantics, for everything except for the sequentially-strict logical operations, we 'just' need to prove that no undesirable crash can occur.

For our example, we have intuitively identified that `self.count > 0` may crash for `invalid` or `null` values of `count`. More rationally, the `self` object is not `invalid` and instance slots cannot contain `invalid` values, so an `invalid` value is only possible as a consequence of a database/network failure. This would be a desirable crash. However the instance slot can contain a `null` value. The `self.count <> null` guard uses `implies` to protect against this `null` value. Activating this protection requires a `false` value for the <> output, enabling us to deduce that the inputs are different.

This achieves what we want, but it has required us to deduce properties of the <> inputs from its output. This is a reverse evaluation that requires distinct implementation programming and which scales badly since only single input monotonic operations support useful deduction of an input from a known output.

We can avoid reverse flow deductions by instead hypothesizing that an undesirable crash can occur and demonstrate that forward evaluation leads to a contradiction. For our example, we consider the hypothesis

execution is attempted for:	self.count > 0
when:	self.count <> null

This evaluation can be performed by a symbolic evaluator that elaborates the standard evaluator to use a symbolic value wherever a constant value is not known.

5.4. Boolean Symbolic Evaluation

For simple invariants, it is sufficient for the *SymbolicValue* to be a tuple maintaining the following information for our partial knowledge:

- Value Type : Type[1]
- MayBeInvalid : Boolean[1]
- MayBeNull : Boolean[1]

The `MayBeInvalid` Boolean distinguishes the two symbolic possibilities: *MayBeInvalid*, *MayNotBeInvalid*. The third *IsInvalid* possibility is a constant literal of type `OclInvalid`.

The `MayBeNull` Boolean distinguishes the two symbolic possibilities: *MayBeNull*, *MayNot-BeNull*. The third *IsNull* possibility is a constant literal of type `OclVoid`.

Resuming our example:

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

The symbolic evaluation needs to confirm that no undesirable crash can occur for the > *OperationCallExp* in `self.count > 0`.

5.4.1. Base Symbolic Evaluation

An overall symbolic evaluation of our invariant uses the initial symbolic value.

Symbolic Variable	Value Type	MayBeInvalid	MayBeNull
self	FixedExample	false	false

and evaluates all other constraints and preconditions

AST element	AST Type	Precondition
self	ParameterVariable	not self.oclIsInvalid() not self.oclIsUndefined() self.oclIsKindOf(FixedExample)
self	VariableExp	
null	NullLiteralExp	
0	IntegerLiteralExp	
self.count	PropertyCallExp	not source.oclIsInvalid() not source.oclIsUndefined() source.oclIsKindOf(FixedExample)
self.count <> null	OperationCallExp	not source.oclIsInvalid() not arg.oclIsInvalid() not source.oclIsUndefined() not arg.oclIsUndefined() source.oclIsKindOf(Integer) arg.oclIsKindOf(Integer)
self.count > 0	OperationCallExp	not source.oclIsInvalid() not arg.oclIsInvalid() not source.oclIsUndefined() not arg.oclIsUndefined() source.oclIsKindOf(Integer) arg.oclIsKindOf(Integer)
self.count <> null implies self.count > 0	OperationCallExp	not source.oclIsInvalid() not arg.oclIsInvalid() not source.oclIsUndefined() not arg.oclIsUndefined() source.oclIsKindOf(Boolean) arg.oclIsKindOf(Boolean)

This gives the symbolic values of each AST node as

AST element	Value Type	MayBeInvalid	MayBeNull
self	FixedExample	false	false
self.count	Integer	false	true
null	OclVoid	false	true
self.count <> null	Boolean	false	false
0	Integer	false	false
self.count > 0	Boolean	true	false
self.count <> null implies self.count > 0	Boolean	true	false

Five of the six `self.count > 0` preconditions are satisfied by the symbolic values.

The sixth, `not source.oclIsUndefined()` might not be satisfied since, for its `self.count` source, *MayBeNull* is `true` and so propagates as *MayBeInvalid* after the comparison.

5.4.2. Hypothesized Symbolic Evaluation

We can establish that the sixth precondition is satisfied by showing that a contradiction results from the hypothesis that the `self.count` can be executed to give a *null* value.

We bind an additional non-symbolic value `null` as the symbolic value of `self.count`.

Symbolic Variable	Value Type	MayBeInvalid	MayBeNull
self.count	OclVoid	false	true

We impose additional preconditions on the short-circuit and if-then-else ancestors of the hypothesized value to ensure that the control path that evaluates the hypothesized value is executable.

AST element	AST Type	Constraint
self.count <> null implies self.count > 0	OperationCallExp	source = true

Re-evaluating the symbolic values of each AST node for the new known values and checking all constraints we find the required contradiction. `self.count <> null` now evaluates to `false` contradicting the new precondition that it is `true` when used as the source of the `implies` operation.

5.4.3. Intuition

Our running example is very simple, closely emulating the simplest of guard idioms that most programmers have used many times. The solution is therefore pretty intuitive.

Laboriously working through the example as symbolic values, constraints, hypothesis and contradiction demonstrates how the magic of intuition and reverse evaluation is replaced by predictable rigor that can scale to non-trivial problems.

Boolean Symbolic Evaluation is sufficient to cope with the complexities of unsafe usage of `null` or `invalid`.

5.5. Real Symbolic Evaluation

Although OCL is not often used for floating point calculations, OCL provides a *Real* type for which division by zero has an undesirable crash hazard.

```
operation Real::/(den : Real) : Real {
    precondition: den <> 0;
}
```

The edit/compile-time analysis should therefore diagnose the rare divide-by-zero hazards.
For the simple case, it is sufficient for a *SymbolicRealValue* to maintain a *MayBeZero* state so that the typical

```
if den <> 0 then num / den else ... endif
```

detects that the divide-by-zero case has been avoided and that the programmer has taken responsibility for solving the problem.
For the general case, it is unlikely that a symbolic analysis can adequately understand non-trivial floating point operations and so the programmer will be forced to adopt the simple-case guard.

5.6. Integer and Collection Symbolic Evaluation

In addition to an *Integer* variant of the rare divide-by-zero hazard, a much more serious hazard arises from a bad index for e.g.

```
aSequence->at(badIndex)
```

The OCL specification provides the preconditions:

```
operation Sequence<T>::at(i : Integer) : T {
    precondition: i > 0;
    precondition: i <= self->size();
}
```

A *SymbolicIntegerValue* needs to track any knowledge regarding the *Maximum* or *Minimum* possible values both as absolute or relative to the `size()` of a base *SymbolicCollectionValue*.

- Actual Type : Type[1]
- MayBeInvalid : Boolean[1]
- MayBeNull : Boolean[1]
- Maximum : Integer[?]
- Minimum : Integer[?]
- MaximumBase : SymbolicCollectionValue[?]
- MinimumBase : SymbolicCollectionValue[?]

The symbolic evaluation of all collection operations needs to relate the output symbolic value to the input so that for e.g. `Sequence::append`, the minimum and maximum output size is one larger than the input size. But for e.g. `OrderedSet::append`, only the maximum output size increases.

5.7. Content Symbolic Evaluation

In addition to tracking the sizes of collections it is also necessary to track known content of collections so that an `includes` guard, `including` action, or `select` filter can satisfy the validity requirements of a subsequent any iteration.

6. Corollaries

Our validity analysis has the goal of guaranteeing that no precondition ever fails. This changes the utility and capabilities of the tooling.

6.1. Preconditions

Without the analysis, a precondition is an additional expression that may cause a crash if evaluated at run-time on actual model values. Since the precondition often just anticipates a crash that would occur anyway, the utility of a precondition is limited to improving the diagnostic that accompanies the crash.

With the analysis, executing preconditions at run-time is redundant. The symbolic execution, at edit/compile-time, on all possible symbolic model values, guarantees that the precondition is satisfied.

Preconditions become an important part of the design and are exploited and checked at edit/compile-time. A too-weak precondition will be diagnosed by a crash hazard within the operation declaring the precondition. A too-strong precondition will be diagnosed by a crash hazard when the operation declaring the precondition is invoked.

6.2. Bodyconditions and BodyExpressions

The Object Constraint Language is actually an expression language so that the functionality of an *Operation* or *Property* is characterized by a body-expression for the respective *ownedBody* or *ownedDefaultValue*. UML only supports constraints and so the `result = bodyexpression` idiom reformulates the arbitrarily-typed body-expression as the Boolean-typed *Constraint*. The UML exposition is indistinguishable from a postcondition.

When specifying OCL, the use of a body-expression is desirable since an implementation may be able to use it for straightforward code generation.

6.3. Postconditions

For an operation such as `sort()`, a postcondition is appropriate to specify the generic characteristics of a bubble or quick sort without imposing any particular implementation.

In addition to the obvious `result = ...` to specify the final result, it is also necessary to provide postconditions for each of the properties of a *SymbolicCollectionValue* such as `result->size() = self->size() + 1`.

Postconditions are never executed by Eclipse OCL. Their execution may be requested in USE. When executed at run-time, they require execution overheads for no benefit, until one fails, at which point a crash must be handled.

Once postconditions form part of the edit/compile-time analysis, many too-weak/too-strong problems may be uncovered in the same way as for preconditions. For library operations at least, a new occasional build-time test could animate each operation with a diverse suite of input values that check the postconditions. For model operations, a similar opportunity exists but work on automated test model generation has revealed challenges.

6.4. Assertions

The new validity analysis benefits from its metamodel focus, but it is never going to be as powerful as a mathematical proof tool and even such specialized tools are unable to prove everything. It is therefore inevitable that a pessimistic validity analysis will have false positives diagnosing non-hazards.

The user will have to provide assistance. An additional invariant or a more explicit guard may often solve the problem. For the harder cases it may be necessary to add an assertion capability.

```
OclAny::oclAssert(constraint : Lambda(T) : Boolean[1],
    justification : String[1]) : OclAny = self
```

The assertion returns its source as its result and asserts that the `constraint` is true for the result[2]. The `constraint` may link to a possibly formal proof of the `constraint` facilitating a QA review of the ad hoc assertions.

The challenge is therefore to make the program flow analysis powerful enough to reduce the number of false positives to a level where the extra user effort to resolve the hazards is more than repaid by the benefits of no crashes.

In many cases, the need for an assertion may alert the user to an unjustified optimism as to the true characteristics of all possible models.

6.5. Static Single Assignment

Symbolic evaluation in OCL is much simpler than in many other languages since OCL is side effect-free, consequently any Common Sub-Expression [1] is immutable and has the same (symbolic) value wherever used. In other languages it may be necessary to refactor to construct a Static Single Assignment representation in which each variable has only a single value. For OCL, all terms are inherently in SSA form.

6.6. Multiple/Cascade Invariants

The constraints for a non-trivial class often comprise some simple obvious constraints and increasingly complicated constraints that depend on the simpler ones.

If each constraint is written in its minimal form and each constraint is checked individually, the tooling is liable to accompany the diagnosis of a simple constraint failure by gratuitous crash diagnoses from the more complicated constraints.

[2]The *Lambda* type is the consequence of modeling the passing of OCL expressions to e.g. iteration bodies.

Conversely, if each constraint is written in its maximal form, the duplication of each simple constraint makes the more complicated constraint hard to read and so leads to maintenance difficulties.

Once we use symbolic evaluation and associate a distinct symbolic value with each distinct AST element, the dilemma goes away. The multiple constraints are part of a logical conjunction for which common sub-expression elimination removes duplicates. Symbolic evaluation observes expression precedence to only traverse credible paths once. Only the first failure in the depth first traversal will be diagnosed.

6.7. Exceptions

Our validity analysis guarantees that no undesirable crashes occur, and that desirable crashes always crash. For the benefit of Operating System level OCL that needs to handle a genuine crash, the ability to use `OclAny::oclIsInvalid()` to catch a crash could be enhanced by an `OclAny::oclAsException()` method to unpack the `invalid` singleton into a new *Exception* class instance for comprehensive handling.

7. Current Status and Further Work

This phase of work was initially driven by the challenges of faithfully implementing undesirable crashes for a QVTc/QVTr Java code generator [14]. The inconveniences of uncrashing logical operations spiral once the logical operations define complex relation guards; an unmatching capability is required. Since the awkwardness of the implementation corresponds quite closely to surprising behavior for the user, it is much more appropriate to push back and alert the user to the hazards and so avoid generating any of the difficult code. For model transformation, the no-undesirable-crashes, all-crashes-always-crash policy is better, simpler and faster.

The work continued the null-safe navigation work [9] which initially required almost all navigation operators to be changed to their safe counterparts; a widespread effort for no real benefit. Extending the null-safe declarations to support null-free collection reduced the changes to genuine hazards. However limited heuristic program control flow analysis meant that only simple guards were recognized as avoiding the hazard.

The new work expands from just null hazards to all precondition failures with a much more comprehensive program flow analysis to propagate, for instance, a known symbolic collection size or content to discount a hazard.

The initial approach of deducing the values of variables backwards from the point at which a guard provides extra knowledge gave way to a forward evaluation of all relevant constraints to contradict hypotheses for each potential hazard.

So far, the prototype demonstrates that an initial symbolic evaluation can associate an overall *MayBeInvalid* and *MayBeNull* state for each (common-)sub-expression. These states can be refined on a local per-expression basis as each symbolic re-evaluation subject to a hypothesis contradicts the *MayBeInvalid* hypothesis. This is currently limited to Boolean Symbolic Evaluation and only a small number of common operations have had their preconditions accurately codified.

Further work is required to codify all operations, support Integer and Content Symbolic Evaluation and to aggregate all applicable invariants to contribute to the overall common-sub-expressions.

Further further work should support auto-generation of all operation-specific code from OCL definitions of their preconditions and bodies.

8. Related Work

Preconditions and postconditions are an essential part of design by contract endorsed by the OCL specification [10], but because they are not much used in practice, related deficiencies in the OCL specification have not been reported. The USE tool [16] is able to execute preconditions and postconditions at run-time. Eclipse OCL [13] never executes them.

The inadequacy of preconditions and postconditions has been highlighted by the proposals to add support for framing conditions [6] that simplistically assert that nothing else changes. It is not clear that framing conditions are necessary for OCL since the prohibition on side-effects ensures nothing changes that isn't mentioned in a postcondition.

Work on automated test model generation [2],[3],[4] produces suites of test models that can be animated. The preconditions can guide the production of good models and motivate the production of bad models.

Tools such as EMFtoCSP [5] and UMLtoCSP establish metamodel consistency by searching for models that demonstrate that all metamodel constraints can be satisfied. An inadequate metamodel is detected if a particular class or constraint is dead with respect to all models searched. The constraints are translated from OCL to a CSP solver where they are repetitively evaluated for candidate models.

These usages rely on converting the (UML) metamodel and OCL constraints into the language of a CSP or SAT solver where searches/syntheses proceed. The precondition is assumed to be internally good and the corollaries are assessed.

In this work, each precondition is proved to be internally good without needing to synthesize any models. Whether the precondition contributes to a usable system is not relevant. We just guarantee that no precondition can ever fail and so remove an impediment to both normal and automated search usage.

9. Conclusions

We have identified the inadequate consideration of crashes as a contribution to the disappointment of many OCL users that constraints are not 2-valued.

We have distinguished between catastrophic desirable crashes and careless undesirable crashes from precondition failures.

We have introduced a compile-time analysis to detect and so guarantee that desirable crashes always crash and that undesirable crashes never occur.

We can look forward to common programming errors such as off-by-one ordered collection index crashes being avoided and so find that constraints are indeed the 2-value construct that users expect.

The prototype implementation within Eclipse OCL shows promise but has revealed how much more can be done.

References

[1] Aho, A., Sethi, R., Ullman, J.: Compilers, Principles, Techniques and Tools, Addison Wesley, 1986

[2] Brucker, A., Krieger, M., Longuet, D., Wolff, B.: A Specification-based Test Case Generation Method for UML. 10th International Workshop on OCL and Textual Modeling, October, 2010, Oslo, Norway. https://modeling-languages.com/events/OCLWorkshop2010/submissions/ocl10_submission_7.pdf

[3] Francisco, M., Castro, L.: Automatic Generation of Test Models and Propertiesfrom UML Models with OCL Constraints. 12th International Workshop on OCL and Textual Modeling, September, 2010, Innsbruck, Austria. https://st.inf.tu-dresden.de/OCL2012/preproceedings/07.pdf

[4] Gogolla, M., Burgueño, L., Vallecillo, A.: Model Finding and Model Completion with USE. 18th International Workshop on OCL and Textual Modeling. Copenhagen, Denmark, October 14, 2018. https://ceur-ws.org/Vol-2245/ocl_paper_9.pdf

[5] Gonzàlez, C., Bũttner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), Jun 2012, Zurich, Switzerland. hal-00688039 https://hal.inria.fr/file/index/docid/688039/filename/emftocsp.pdf

[6] Przigoda, N., Filho, J., Niemann, P., Wille, R.: Frame Conditions in Symbolic Representations of UML/OCL Models. 2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), 2016, pp. 65-70, doi: 10.1109/MEMCOD.2016.7797747. https://iic.jku.at/files/eda/2016_memocode_frame_conditions_symbolic_representation.pdf

[7] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley, Jun 2008

[8] Willink, E.: Reflections on OCL 2. Journal of Object Technology. Vol.19, No. 3, 2020. https://dx.doi.org/10.5381/jot.2020.19.3.a17

[9] Willink, E.: Safe Navigation in OCL. 15th International Workshop on OCL and Textual Modeling, September 8, 2015, Ottawa, Canada. https://ocl2015.lri.fr/OCL_2015_paper_1111_1400.pdf

[10] Object Constraint Language. Version 2.4., OMG Document Number: formal/2014-02-03, Object Management Group. February 2014, https://www.omg.org/spec/OCL/2.4

[11] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. OMG Document Number: ptc/16-06-03, June 2016. https://www.omg.org/spec/QVT/1.3

[12] OMG Unified Modeling Language (OMG UML), Version 2.5, OMG Document Number: formal/15-03-01, Object Management Group, March 2015, https://www.omg.org/spec/UML/2.5

[13] Eclipse OCL Project. https://projects.eclipse.org/projects/modeling.mdt.ocl

[14] Eclipse QVT Declarative Project.
https://projects.eclipse.org/projects/modeling.mmt.qvtd

[15] OCLinEcore. https://wiki.eclipse.org/OCL/OCLinEcore

[16] USE, The UML-based Specification Environment.
https://useocl.sourceforge.net/w/index.php/Main_Page