Benefits of type constraints over OCL constraints

Matthias Beyer and Julian Ganz and Peter Wursthorn
Faculty of Computer Science
Hochschule Furtwangen
Robert-Gerwig-Platz 1
78120 Furtwangen

***Abstract*—The Object Constraint Language (OCL) is used to annotate the code of a program with constraints that must be resolved by a resolution engine. Another way of defining declarative constraints on a program is by using types annotations. In this paper, we show that a transformation mapping OCL constraints to type system constraints can be constructed for type systems featuring "generics" or "templates". We also discuss some constraints which cannot be expressed in OCL but are commonly realized using types. Hence, we show that using a type system, more constraints can be expressed than with OCL. This, to some degree, contradicts the use of OCL for software modeling.**

I. Introduction

Object Constraint Language (OCL) is a declarative language for source code constraint annotations that is used to extend the specifications which are expressed via Unified Modeling Language (UML). OCL is used to annotate the code of a program with constraints that must be resolved by a resolution engine. This can be done statically or dynamically at runtime of the program. In the first case, the resolution engine must be able to understand the source code of the program to be able to execute the annotated constraints. In the second case, the program needs to be able to execute the constraints and handle failure appropriately.

Another way of defining declarative constraints on a program is by using types annotations, if the used programming language features types.

In this paper, we briefly discuss both OCL and type systems. We show that type annotations can be used to express the same results as one could express with OCL annotations, in some existing type systems. This is done by showing that a transformation exists which maps any OCL expression to a constraint realized through the type system. We discuss enforcing those constraints both during compile- and run-time.

Furthermore, we discuss some constraints which are impossible to express in OCL but commonly realized using types. We show that, concerning constraints, type systems are more powerful than OCL. The evaluation is accompanied by the discussion of difficulties of techniques outlined.

II. Related work

Both type systems and OCL are broadly discussed in the literature. For those systems, publications exist discussing both theoretical properties and practical implications.

For example, a blog post from 2010 [1] provides a brief overview over discussion points regarding type systems as well as common misconceptions. The post appears to be targeted at computer programmers in general, especially ones with little experience. Discussion points are not backed by either literature or formalism.

The book "Types and programming languages" by Pierce [2] discusses type systems in some depth for a broad target audience. It covers core topics and many properties of type systems as well as techniques relevant for realizing and analyzing type systems. The practical use of type systems in programming is one of the main goals of the book [2, p. xiii f.]. Pierce also authored a second book "Advanced topics in types and programming languages" [3].

Papers such as "Type systems" [4] discuss type systems from a purely theoretical perspective. To avoid ambiguities, Cardelli introduced precise definitions of some terms which are commonly used when discussing type system. Also, he introduced a formalism, which appears to be derived from formal and predicate logic.

Papers which discuss single topics, e.g. categories of constraints, also exist. Aiken and Wimmers [5], for example authored a paper on "Type Inclusion Constraints and Type Inference". Also, relatively recent development of type systems can be observed. For example, the concept of "traits", which are used in the Rust programming language, were presented by Schärli et al. in 2003 [6].

Apparently, OCL is not in the same focus of research that type systems are. The primary source for OCL is its specification [7]. However, many secondary literature such as educative materials, manuals, howtos and guides exist for OCL. The collection "Object Modeling and OCL" [8] and the paper "On the expressive power of OCL" [9] by Mandel and Cengarle are examples of more academic works on OCL.

III. Type systems

There is no uniform definition of the term *type system*. Pierce defines the term *type system* as

> a traceable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [2]

Pierce notes that there are two kinds of type systems in research - the practical one, which concerns applications to programming languages and, the more abstract one, pure typed lambda calculi [2].

While programming languages (and hence type systems) can be categorized in either *typed* and *untyped* languages [4, p. 2 f.], they cannot be categorized in *strong* and *weak* type systems. Although, a order can be established, where type systems are compared to each other [1].

Type systems are a form of constraints which can be imposed on a program. If the constraints hold, the program is *well typed* and hence will not fail during runtime[1] [5].

> The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program. [4, p. 1]

We consider the type systems of C++ and the Rust programming language modern type systems, amongst others.

A. Rust

Rust is a practical, (memory-)safe, statically typed, imperative programming language [10] that features generics, traits and type classes [6], [11]. Rust also features *move semantics* like C++, pattern matching, type inference, zero-cost abstractions, has no garbage collector and prevents iterator invalidation and data-races [10], [11]. As Rust has a substructural, affine type system [3], [10], a resource can only be used once, while in a linear type system, a resource must be used once. An example introduced by Pierce in [3] explains, that with a substructural type system, one can specify that a file must be opened before read, and may never be read after it is closed. Thus, the order of actions on values can be controlled.

> Substructural type systems are particularly useful for constraining interfaces that provide access to system resources such as files, locks and memory. Each of these resources undergoes a series of changes of state throughout its lifetime. Files, as we've seen, may be open or closed; locks may be held or not; and memory may be allocated or deallocated. Substructural type systems provide sound static mechanisms for keeping track of just these sorts of state changes and preventing operations on objects in an invalid state. [3, p. 3 f.]

For example, in the Rust programming language it is not possible to write to a file that is closed. The code in Listing 1 does not compile because of the last function call in the `main` function; the compiler is able to recognize that the variable `f` is already dropped and therefor not longer accessible, so calling `write()` on it is not possible.

Now consider the same application in the Java programming language, as shown in Listing 2. The java compiler compiles the code, though running of the application results in a backtrace with an exception thrown.

Because of the stricter type system Rust features (compared to Java or C for example), it is possible to prevent mutable sharing of resources to multiple users, which is not possible in languages with weaker type systems.

[1] The stronger the typing of the programming language is, the stronger is this guarantee. For example, a Java program might be well typed but fail anyways during runtime while an equivalent program written in the Haskell programming language will not fail, as the type system of the haskell programming language is stronger than the one of the Java programming language.

Listing 1: Rust: Invalid file usage after close

```
fn main() {
    let mut f = File::open("/tmp/example")
        .unwrap();
    let _ = f.write("example");
    drop(f);
    f.write("doesn't work");
}
```

For example, the compilation of Listing 3 fails: the compiler reports that "f" cannot be captured by the second closure since it was already "moved" to the first thread through the corresponding closure.

Now consider the same application in a less strictly typed language, where mutable sharing of resources is allowed by the compiler or interpreter. It is undecidable which text is written to the file first, because mutable sharing of resources is not forbidden.

B. Type safety

The term *Type safety* describes the safety measurements one can establish via a type system. The strength of a type system, as described previously in this section, actually describes how *safe* the constraints on the program are, whereas *safe* means how much they are enforced by the compiler.

The `printVehicle` method shown in Listing 4 for example takes an object of type "Vehicle". Thus, it cannot be called with, for example, a "String". Although, it can be called with a "Car", which extends "Vehicle".

The same function in Ruby (as shown in Listing 5) can be called with either "Vehicle" and "Car". Although another type (for example "String" or "Array") would work, it might produce an runtime error because the operations applied inside the function are not defined for these types.

We can see from Listing 4 and Listing 5 that Ruby is *less type safe* than Java, as types are not enforced in Ruby. This is a feature of *dynamically typed* languages and may be beneficial in some use cases.

In non-object-oriented languages, the above examples could be reproduced in a slightly different manner. For example, in

Listing 2: Java: Invalid file usage after close

```
FileWriter fw =
    new FileWriter("/tmp/example");
BufferedWriter bw =
    new BufferedWriter(fw);
bw.write("example");
bw.close(); fw.close();
bw.write("example");
```

The code example is shrinked to fit the size of the paper. Consider all necessary packages be imported and the code embedded in a class with a main function.

Rust or Haskell, *traits* and *type classes* are used to abstract over types. So, one can define a *trait* which defines an interface for a type. A function may take a parameter of a type which implements this *trait*, but no other type (as shown in Listing 6).

In Listing 6 a *trait* "T" is defined which requires the implementation to have a function "p". Both types "A" and "B" implement this *trait*, and print something to the standard output. A function "print_this" is defined which takes a reference to something that implements the *trait* "T" and calls its function. The actual *type* of what is passed into the function is irrelevant, the constraint only specifies that the function takes everything which implements the *trait*[2].

Besides the points shown above, statically and "strongly" typed languages feature more benefits. For example, as studies have shown, statically typed languages improve maintainability of applications [12], [13].

IV. OCL

OCL is a declarative language for describing constraints, which is part of the UML language. It first appeared to overcome the shortcomings of UML and quickly extended its scope to become one of the key components of Model Driven Engineering (MDE) [14].

OCL is a *side effect free* [7, p. 5], typed, declarative language. Especially the absence of *side effects* is beneficial, as the constraints typed out in OCL can query and process system state, but not modify it. Contracts for objects can be defined as *invariants*, *pre-* and *post*conditions that restrict the value of an object [14]. In addition to the language itself, the OCL specification also declares some points of compliance for OCL tools [7, p. 1].

[2] This can also be done in Haskell with *type classes*, which serve the same purpose.

Listing 3: Rust: Shared mutable state

```rust
fn main() {
    let mut f = File::open("/tmp/example")
        .unwrap();
    let t1 = thread::spawn(move || {
        let _ = f.write(b"foo");
    });
    let t2 = thread::spawn(move || {
        let _ = f.write(b"bar");
    });
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Listing 4: Java: Vehicle and Car

```java
class Vehicle { /* ... */ }
class Car extends Vehicle { /* ... */ }

void printVehicle(Vehicle v) { /*...*/ }
```

OCL has access to the current execution context, thus it can read available variables and also traverse member variables of objects (see Listing 7).

OCL itself has a type system. A value in OCL can be either an atomic type or a template type. Atomic types can be either predefined types ("Integer", "Real", "String" or "Boolean") or user-defined ones, which are either class types or enumeration types. A template type is a generic *Collection* type. Overall, OCL values can be sorted in nine categories of types [14].

The OCL standard also defines a number of boolean- and set-operations for the types OCL offers, including type conversions and iteration functionality for the complex types.

V. Transformation from OCL constraints to type system constraints

If OCL constraints can be replaced by type system constraints, a transformation exists which transforms an arbitrary OCL constraint to a type system constraint. In this section, we show how such a transformation can be constructed as well as how the constraints can be applied.

A. Representation of constraints as types

The concept of "templates" or "generics" allows parametrization of types (e.g. classes). In C++, for example, a developer may specify class templates. A class template specifies a generic structure which may be partially based on parameters rather than concrete types or values [15]. In this context the term "instantiation" refers to the "concrete" type[3], which is yielded if all parameters are specified for a template.

It is possible to use instantiations of a template for other template instantiations.

Lemma 1. *This property allows constructing* types *representing arbitrary data or information.*

Proof. Consider the two templates shown in Listing 8. Using these two templates, one may declare chains of `zeros` and `ones`, with each struct referring to the next element through its `Next` parameter, terminated by the **`void`** type.

Using such a chain, one may represent arbitrary bit sequences of arbitrary length[4] and thus arbitrary data or information. □

[3] e.g. suitable for declaration of run time variables or constants

[4] The length one is be able to use in practice may depend on the compiler as well as the computer performing the compilation.

Listing 5: Ruby: Vehicle and Car

```ruby
class Vehicle; end
class Car < Vehicle; end

def print_vehicle v
    # ...
end
```

As OCL constraints can be viewed as information, they may also be represented as types. Of course, embedding OCL constraints as bit sequences is rather unpractical. In practice, one would rather represent an OCL expression through its Abstract Syntax Tree (AST) using templates each representing one node of the AST. A more practical representation is discussed in subsection V-C.

Also important for the discussion is what can be done using the representation of an OCL expression as a type. The mere representation of an expression is of little use if the expression cannot be evaluated.

B. Static evaluation

It has been shown that arbitrary computation can be performed using Template Metaprogramming (TMP) in C++ [16]. Thus, one can process information embedded in types at compile time and use the result at run time or to abort the compilation if a condition is not met[5]. The same is possible in Rust [17] and Haskell [18] using similar techniques.

Lemma 2. *Any valid OCL expression may be evaluated at compile time given the values used in the expression.*

Proof. It was previously stated that anything that can be computed can be computed at compile time using TMP or a similar technique. Evaluating an OCL expression is a decidable problem. Therefore, the result of an expression can be computed. Hence, it can be evaluated at compile time. □

Furthermore, it is both possible and practical to refer to the context of an OCL expression in the source code using template parameters. This enables additional validation of the OCL expression. For example, one may enforce compliance of the attributes to a method stated in an OCL expression with the attributes declared for a method.

More interesting than merely evaluating OCL expressions might be analyzing a set of constraints concerning their soundness. If a contradiction is found in a set of expressions, a redesign may be necessary. A resolution engine is already used in existing OCL tools in order to uncover contradictions in sets of OCL expressions. Such an engine may, conceptually, also be implemented using TMP, enabling compile time validation of an OCL specification.

However, there are limits to what can be achieved using this kind of static analysis at compile time. The statements and control flow of a method are not exposed through the type system in languages like C++ or Rust and are thus not accessible to a static analyzer implemented using TMP.

Even if a static analyzer has access to the implementation of a method, the correctness and conformance to constraints can not generally be proofed statically in every case. Hence, static analysis may be complemented by run time enforcement of constraints.

C. Evaluation at run time

Conceptually, it is also possible to evaluate an OCL expression at run time, if all the values used in the expression are available. The aforementioned type representation of an OCL expression may be used or extended for this purpose.

Furthermore, C++ allows specifying pointers to functions and methods as well as attribute offsets. This allows crafting structures which are capable of accessing methods and attributes of objects, given an instance[6]. Alternatively, it is possible to construct method or attribute "accessors" manually or using some sort of closures. Conceptually, the same is possible in Rust and other languages.

Lemma 3. *From a type representation of any well-formed OCL expression, it is possible to construct a function or functor which evaluates the expression, given all values used in that expression.*

Proof. We already established that, given the AST of an expression, it is possible to construct arbitrary output from it using TMP, where the output is a type (lemma 2). This type may be an arbitrarily complex template. We also established that it is possible to refer to attributes and methods of objects using templates and hence invoke them at run time based on type information.

[5] e.g. using `static_assert` in modern C++.

[6] For example, the C++ Meta Object Helpers (CMOH) project[19] aims at providing some arbitration based on such mechanisms

Listing 6: Rust: Passing by Trait

```
pub trait T { fn p(&self); }
fn print_this<P: T>(p: &P)
{ p.p(); }

struct A(pub i32);
impl T for A { fn p(&self) { /*...*/ } }

struct B(pub String);
impl T for B { fn p(&self) { /*...*/ } }

fn main() {
    let a: A = A(12);
    let b: B = B(String::from("Hello"));
    print_this(&a);
    print_this(&b);
}
```

Listing 7: OCL constraint: The number of passengers may never exceed the number of seats in a car

```
context Car
inv: self.passengers->size() <=
 ↪  self.seats->size()
```

Listing 8: Types representing bits in a bitstring

```
template <typename Next> struct zero {};
template <typename Next> struct one {};
```

Using function overloading and/or varying implementation of a method for a set of types, one can construct arbitrary functions of arbitrary complexity which are executed at run time. Hence, it is possible to construct a function which evaluates an OCL expression for a list of values containing the values used in the expression. □

Remark 4. Note that values postfixed with `@pre` are separate values in this context.

In the following paragraphs, we want to line out a practical realization for such a run time evaluation.

Consider a function template or equivalent which takes an arbitrary number of arguments and returns one specific argument based on some static selection specifier. Such a function template may be used to evaluate (sub-)expressions which only consist of a single value, if the argument list contains the value used in the expression. Therefore, this function template or class of function will be called "value function" in this section. Such a class of functions is generally possible because OCL expressions are, by definition, free of side-effects [8]. Hence, values can be passed as immutable references or by value, e.g. without "moving" or manipulating it, depending on the concrete language.

Now consider a representation of an OCL expression where each sub-expression is represented as an instantiation of a template specific for the expression class as described by OCL expressions package. The template takes the instantiations corresponding to the sub-expressions as template parameters. Hence, each instantiation does not only represent the node, but the expression for which the node is the root node. For this reason, we will henceforth refer to those templates as "sub-expression" templates.

Like "value functions", these templates may either be function templates or structs or classes exporting an "evaluation" method. In the role of a function, each "sub-expression" template takes a number of arguments containing all values used in the expression. Those values are forwarded to the templates representing the sub-expressions. If we consider "value functions" also "sub-expression" templates, an OCL expression can be evaluated at run time by calling the "sub-expression" template of the root node with the values used in the expression.

D. Enforcing constraints at run time

Any function constructed for the purpose of evaluating an OCL expression is, naturally, only of use if it is called. Explicit calls to functions evaluating OCL expressions could be called explicitly. However, at least in some languages, it should also be possible to provide a generic wrapper with which any call to a function or method may be wrapped. The wrapper would call the function or method and assert the conditions for the context.

The wrapper has access to all values which may occur in expressions, since it may access any parameter passed to a function or method as well as return values[7]. Potentially, uses of the `@pre` post-fix could be problematic if the value affected by it is "consumed" or modified by the function or method. Copying or cloning the value for the purpose of evaluating post-conditions may also be illegal.

There are a number of options of how to handle violated conditions, such as throwing an exception or aborting the program. It is also possible to return a "result" or "option" type, which only contains a value if all conditions are met.

Using such a framework in practice in production code might be cumbersome but still useful for writing tests. Theoretically, a compiler plugin of some sort could be used to insert or extend the necessary function calls. However, the purpose of this section is merely to show that an OCL expression can be represented as a type which allows evaluation of the original expression preserving the semantics. Hence, we will not concern ourselves further with triggering the evaluation.

E. Example: Boolean expressions

As a demonstration of the techniques discussed in this section, we developed a minimal example in C++14. We declared templates which may be used to represent arbitrary Boolean expressions. More precisely, we declared a template to represent conjunctions, disjunctions, negations and enumerated values.

Except for the latter, the templates take the subexpression(s) on which to apply the operator as type parameters. The value template instead takes an unsigned number, through which different values in the expression may be differentiated. Additionally, each of the "struct"-templates declare a member "eval" which evaluates the associated subexpression for a list of supplied values at runtime. The evaluation is done recursively, as described in subsection V-C.

Listing 9 illustrates the declaration of the conjunction operator as a template. Listing 10 illustrates the "value" template, including the mechanism used for selecting a specific parameter from the method arguments.

[7]In the case of methods, the object on which the method is invoked may be regarded an argument. In some programming languages (e.g. Python, Rust), this is made explicit, syntactically.

Listing 9: Conjunction

```
template <typename L, typename R>
struct conjunction {
    typedef L l;
    typedef R r;

    template <typename... A>
    static constexpr auto
    eval(A const&... a) {
        return L::eval(a...) &&
         ↪ R::eval(a...);
    }
};
```

It is also possible to check whether an expression constructed from the templates discussed above is satisfiable. To achieve this, we first transform the expression into Disjunctive Normal Form (DNF) using a series of substitutions. The first set of those substitutions is illustrated as a recursive transformation in Equation 1. The purpose is the "atomization" of an expression e. The atomized expression is equivalent to the original. However, all negations in the atomized expression act only on single values rather than on conjunctions or disjunctions.

$$a(e) = \begin{cases} a(x) \wedge a(y) & \text{if } e = x \wedge y, \\ a(x) \vee a(y) & \text{if } e = x \vee y, \\ \neg a(x) \vee \neg a(y) & \text{if } e = \neg(x \wedge y), \\ \neg a(x) \wedge \neg a(y) & \text{if } e = \neg(x \vee y), \\ a(x) & \text{if } e = \neg\neg x. \end{cases} \tag{1}$$

Using a similar recursive transformation, the expression is transformed in a DNF by expanding all conjunctions of a disjunction and another expression, e.g. as illustrated in Equation 2. In the resulting expression, a conjunction may appear as a subexpression to a disjunction, but a disjunction will not appear as the subexpression of a conjunction. Therefore, the expression is in DNF.

$$x \wedge (y \vee z) \mapsto (x \wedge y) \vee (x \wedge z) \tag{2}$$

For applying the substitutions, we leverage the ability of a modern C++ compiler to match a type against a parameter in a template specialization in order to select a substitution. This is possible because the compiler automatically selects the most specialized variation of a template for instantiation. Listing 11 demonstrates this technique for a single substitution used for atomization.

A Boolean expression in DNF is satisfiable if at least one of the conjunction it contains is satisfiable. A conjunction in turn is satisfiable if it doesn't contain both a value and its negation. After the expression is transformed to DNF, the satisfiability of the expression can be determined by: (1) recursively checking whether at least one subexpression of a disjunction is satisfiable; (2) linearizing the conjunctions by recursively moving sub-conjunctions from the left to a right for each conjunction; and (3) check recursively whether the right hand side of a conjunction contains the negation of the expression on the left hand side for each (linearized) conjunction.

As illustrated in Listing 12, the result of each single check is returned as a enumeration value, which is a compile time constant. It can then be used in a `static_assert()` in order to generate a compile time error, should the expression not be satisfiable.

F. Limitations

While the limits of the static verification lie in what can be refuted using an resolution engine, the limits of dynamic evaluation are of more practical nature.

We have shown that an expression can be evaluated given all the values used in the expression. However, not all values may be available in a certain program context. Also, not all concepts described in the OCL specification apply to programming languages referred to in this paper. This supposedly led to some points regarding evaluation be optional regarding the conformance of OCL tools. Hence, we use this list of optional features to discuss the limitations of the dynamic evaluation described above:

1) `allInstances()`*:* This function returns a set referring to all existing instances of a type. It is problematic for multiple reasons. Generally, a program-wide registry of objects with associated types does not exists. It may be possible to create such a registry using a custom allocator. However, in languages like C++ and Rust, objects are often kept on the stack or inside the memory area of other objects. Hence, it may be necessary to create such a registry manually or by other means.

Furthermore, even if the set contains (immutable) references, accessing objects within it can still be problematic in

Listing 10: Value

```
template <unsigned int num>
struct value {
    template <typename A0, typename... A>
    static constexpr auto
    eval(A0 const& a0, A const&... a) {
        return value<num - 1>::eval(a...);
    }
};
template <>
struct value<0> {
    template <typename A0, typename... A>
    static constexpr auto
    eval(A0 const& a0, A const&... a) {
        return a0;
    }
};
```

Listing 11: Atomization

```
template <typename T>
struct atomized {
    using type = T;
};
template <typename L, typename R>
struct atomized<negation<conjunction<L,
 ↪ R>>> {
    using type = disjunction<
        typename
         ↪ atomized<negation<L>>::type,
        typename
         ↪ atomized<negation<R>>::type
    >;
};
```

multi-threaded environments, at least in non-garbage-collected languages[8].

2) `@pre` in postconditions: As already pointed out in subsection V-D, using the `@pre`-postfix could be problematic if the value it refers to was "consumed" or modified during a function call.

3) `OclMessage`: This concept does not generally apply to programming languages described in this paper. Hence, application would have to be evaluated for the specific language and/or framework introducing this concept.

4) Navigating non-navigable associations: This is also highly problematic because the data necessary to track back a reference does not generally exist. Furthermore, it may often not be possible to *address* the non-navigable association from a context, since we only have access to the named references which are part of the context. Usually, the means of addressing such associations is provided by an accompanying UML document, which is lacking in this context.

5) Accessing private and protected features: If we use the type system as well as the programming language itself to craft evaluation functions, we are also restricted regarding the visibility of a type's features. Hence, we cannot access those features which are private or protected unless we prepare the affected type to allow the necessary access from our evaluation framework.

The conformance points discussed above are not specific to the constraints yielded by the transformation but to the programming language or runtime. They may apply to various OCL tools. Therefore, those limitation do not restrict the generality of possible transformations which can be constructed.

However, the construction technique relies on the use of generics or templates. Hence, it can only be applied to type systems with an equivalent feature.

VI. Other type system constrains

As the name suggests, the Object Constraint Language (OCL) is a language designed to declare constraints on *objects*.

This naturally restricts the application of OCL. For example, OCL allows using free functions in an expression (e.g. invariant or pre-condition) via a "operation call expression". However, a free function cannot be specified as a "context" syntactically, since a context declaration in OCL requires an operation to be prefixed with a path name [7, p. 194] which, in turn, cannot be empty [7, p. 75]. It is therefore not possible to specify any constraint on a free function using OCL, while the compiler naturally enforces the constraints imposed by the types involved in the operation.

[8]since an object might be destructed between the retrieval of the set and the access of the specific object

Listing 12: General template for determination of satisfiability

```
template <typename T>
struct dnf_is_satisfyable {
    enum : bool { value = true };
};
```

One could argue that a function should always be the member of a namespace or package. Also, it is possible to work around this limitation by defining an alias for an empty package path or the root package or namespace. However, this syntactical predicate already demonstrates that OCL is limited due to the focus on Object Oriented Programming (OOP).

A. Higher order functions

The lack of an expressive tool such as "inline" or anonymous contexts becomes more relevant when higher order functions are involved. A higher order function in this context is a function (or method) which takes as an argument at least one function or functor [20, p. 154]. The `fold()` function in Listing 14, for example, is a higher order functoin.

In OCL, it is possible to represent constraints on a function type through a class having a single operation representing the "call" to the function. In the context of the higher order function receiving that function pointer or callable object, we can refer to the class associated with the callable. This, however, also requires the application of additional conventions which are not part of the OCL specification.

While the workaround described may resolve issues regarding syntactic limitations of OCL, it does not lift *semantic* limitations regarding function types. For example, consider the predefined OCL properties `oclIsTypeOf(t)` and `oclIsKindOf(t)`. The specification states that

> The operation is *oclIsTypeOf* results in true if the type of self and *t* are the same. [7, p. 23]

and

> The *oclIsKindOf* property determines whether *t* is either the direct type or one of the supertypes of an object. [7, p. 23]

Other means for determining or matching the type of an object are not specified by the OCL specification.

Those properties cannot be generally used for identifying function types in OCL expressions, e.g. for accepting a set of functions satisfying one of several combinations of constraints. Using those properties would imply that only functions of that specific *named* type, satisfying the constrains declared on it are accepted. However, explicitly declaring the type by name on functions is neither practical nor generally possible. Consider, for example, lambda expressions or closures.

Rather, a software engineer may wish to accept any function which satisfies a set of constraints without the requirement of explicitly declaring a name on the type of the specific function used. Existing type systems allow declarations of such constraints, e.g. through the argument types and the return value's type associated to the function type.

B. Templates and Generics

The disability to express constraints on unnamed types becomes also relevant when considering templates and generics.

The constraints which can be expressed for type parameters in Java generics let the developer enforce that the type extends or implements a named class or interface. Similar constraints

can be expressed in Rust generics. It is possible to map those expressions to OCL expressions using the `oclIsKindOf()` property discussed above in most cases[9].

However, the constraints possible for template parameters are far more arbitrary in languages like C++. For example, simply by using an attribute or operation in a template enforces the presence of the property for the type supplied in an instantiation. It is also possible to enforce the presence of attributes or operations for a type more gracefully using the Substitution Failure Is Not An Error (SFINAE) technique [21, sec. 23.5.3] (refer to Listing 13 for an example). Besides enforcing the mere presence of an operation[10], the technique also allows enforcing some properties of those operations. While some named abstractions exist for commonly used "type traits", it is neither necessary nor practical to name constraints in all situations.

Listing 13: Demonstration of the SFINAE technique

```
template<typename T, typename = void>
struct printable_vec;

template<typename T>
struct printable_vec<
    T,
    void_t<
        decltype(std::declval<T>().p())
    >
> : public std::vector<T> {
public:
    void print() { /* print elements */ }
};

struct foo { public: void p(); };

int main() {
    printable_vec<foo> a;
    // ok, specialization used
    printable_vec<int> b;
    // err, general version used,
    // which is an incomplete type
}
```

Additionally, recent C++ standards allow declaration of templates which take a variable number of template arguments. These templates are called "variadic templates". Using variadic templates, it is possible to declare type safe functions taking a variable number of arguments [21, sec. 3.4.4]. Hence, it is possible to enforce arbitrary constraints on each of the arguments. Furthermore, it is possible to express constraints across multiple arguments.

It appears that it is not possible to express such a function in OCL, syntactically. It is therefore not possible to express constraints on these functions in OCL while it is possible to express them in the C++ type system.

C. Immutability and object lifetime

"Constness" or mutability is one of the most basic properties of a variable's type in C++, Rust and Haskell, which is enforced by the compiler [21, sec. 2.2.3]. In this context, a "const" or immutable value cannot be altered or modified. Successive reads of some attribute from a "const" object will yield identical results, e.g. it is a valid optimization to read the attribute only once. Also, it is possible to specify that an operation does not alter a value, e.g. the value is immutable in the context of the operation. Listing 14 illustrates this class of contraints.

Listing 14: Rust iterator example

```
fn concat(mut acc: String, word: &&str)
    -> String
{
    acc.push_str(word);
    acc.push_str(" ");
    acc
}

fn main() {
    let v = vec!["Hello", "friends"];
    let s = v.iter()
        .fold(String::new(), concat);
    println!("{} {}", s, v.len());
}
```

This example demonstrates both immutability and higher order functions. The `concat()` function is used to concatenate strings in a *mutable* accumulator. The words are, however, immutable in the context of the function. The immutability is required by the main function. The `iter()` function returns an iterator over immutable items. Hence, they cannot be altered by the `fold()`.

OCL does not provide any means to express the mutability of an object, value or context nor is the concept described anywhere in the specification. The term "constant" only appears in contexts discussing constant expressions, e.g. literals. This is somewhat surprising when considering that a core property of expressions required from OCL expressions is that they are free of side effects. On the other hand, the authors gained the impression that OCL is very much influenced by the Java community. In this domain, the concept of immutability appears to be available for references, at best.

Another concept which can't be expressed in OCL is the ownership of an object. In C++ and Rust an operation may "consume" an object, making it unavailable in the calling context after the call. While in C++, the "move" is an operation which is defined semantically and implemented using a type of reference, consuming parameters is directly expressed via the type system in Rust. In C++, the "value" associated to an object is moved but the object itself remains in place, possibly allowing further use, e.g. assigning a new value [21,

[9]However, as discussed above, closures may still be problematic.

[10]e.g. a method, an overload of a free function or method of another class

sec. 3.3.2]. In Rust, however, moving a value invalidates the associated binding, making programs attempting to use the binding afterwards ill-formed.

Rust also allows expressing the lifetime constraints of references and other dependents on an object. Needless to say, it is not possible to express such constraints in OCL, at least not without heavy syntactic or semantic modifications.

VII. Conclusion

In section V the set of constraints expressible via OCL C_{OCL} is a subset of the constraints realizable through a type system C_{TS} featuring "templates" or "generics".

$$C_{OCL} \subseteq C_{TS} \tag{3}$$

As described in subsection V-B, static evaluation is possible if those "templates" or "generics" are turing complete. Those requirements on type systems apply for the specific approach taken for constructing a transformation. It may well be possible to construct other transformations for type systems not featuring generics.

Readers should note that this paper makes use of templating techniques to express and realize constraints in order to discuss the construction of a transformation. Type systems typically offer more direct means of expressing an constraint.

In section VI, we identified constraints which generally can not be expressed via OCL:

$$C_{OCL} \neq C_{TS} \tag{4}$$

We thus showed that

$$C_{OCL} \subset C_{TS} \tag{5}$$

Therefore, OCL annotations may not provide any additional value compared to "bare" declarations of types and methods in some modern programming languages. A "bare" declaration in this context is a declaration, e.g. of a function, which only provides type information but contains no implementation.

This, to some degree, contradicts the use of OCL for software modeling. For example, instead of extending an UML class diagram with OCL expressions, software engineers could use type annotations of the target language and achieve a similar result. While UML tools may not be able to perform a static analysis based on type annotations, the constraints expressed using type annotations are effective in the implementation, without the necessity of additional tools or code generators.

The power of modern type systems also has implications regarding interface design. Instead of documenting constraints on functions and methods of an Application Programming Interface (API) in OCL or plain text, interface designers should rather leverage the capabilities of the language's type system.

For example, if an integer value cannot, conceptually, be negative, interface designers should use unsigned variants of integer types. Qt's `QVector::resize()` method [22] can be considered a negative-example for this.

Acknowledgment

The authors would like to thank Kai Sickeler for his extensive review of this paper.

References

[1] "What to know before debating type systems," 2010. [Online]. Available: http://blogs.perl.org/users/ovid/2010/08/what-to-know-before-debating-type-systems.html

[2] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[3] ——, *Advanced topics in types and programming languages*. MIT press, 2005.

[4] L. Cardelli, "Type systems," *ACM Computing Surveys*, vol. 28, no. 1, pp. 263–264, 1996. [Online]. Available: http://www.cs.colorado.edu/~bec/courses/csci5535/reading/cardelli-typesystems.pdf

[5] A. Aiken and E. L. Wimmers, "Type inclusion constraints and type inference," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93. New York, NY, USA: ACM, 1993, pp. 31–41. [Online]. Available: http://doi.acm.org/10.1145/165180.165188

[6] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behaviour," in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 248–274. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-45070-2_12

[7] *Information technology - Object Management Group Object Constraint Language (OCL)*, ISO/IEC JTC1 and OMG, February 2014.

[8] T. Clark and J. Warmer, Eds., *Object Modeling with the OCL*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. [Online]. Available: https://link.springer.com/book/10.1007/3-540-45669-4

[9] L. Mandel and M. V. Cengarle, "On the expressive power of ocl," in *International Symposium on Formal Methods*. Springer, 1999, pp. 854–874. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-48119-2_47

[10] A. Beingessner, "You can't spell trust without rust," Ph.D. dissertation, Master's thesis, Carleton University, 2015.

[11] A. Light, "Reenix: Implementing a unix-like operating system in rust," Ph.D. dissertation, Master's thesis, Brown University, Department of Computer Science, 2015.

[12] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 153–162. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6240483/

[13] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, 2014. [Online]. Available: https://link.springer.com/article/10.1007/s10664-013-9289-1

[14] J. Cabot and M. Gogolla, "Object constraint language (ocl): a definitive guide," in *Formal methods for model-driven engineering*. Springer, 2012, pp. 58–90.

[15] J. Siek and W. Taha, *A Semantic Analysis of C++ Templates*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 304–327. [Online]. Available: http://dx.doi.org/10.1007/11785477_19

[16] T. L. Veldhuizen, "C++ templates are turing complete," *Available at citeseer. ist. psu. edu/581150. html*, 2003. [Online]. Available: https://pdfs.semanticscholar.org/55a1/417c034899636e736cfb168071555641dece.pdf

[17] S. Leffler, "Rust's type system is turing-complete." [Online]. Available: https://sdleffler.github.io/RustTypeSystemTuringComplete/

[18] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 2002, pp. 1–16. [Online]. Available: https://dl.acm.org/citation.cfm?id=581691

[19] J. Ganz, "Cmoh: C++ meta object helpers." [Online]. Available: https://github.com/neithernut/cmoh

[20] P. B. Andrews, *An introduction to mathematical logic and type theory: to truth through proof*. Orlando: Acad. Press, 1986.

[21] B. Stroustrup, *The C++ programming language*, 4th ed. Addison-Wesley Professional, 2013. [Online]. Available: http://proquest.tech.safaribooksonline.de/9780133522884

[22] The Qt Company Ltd. (2017) QVector Class. [Online]. Available: https://doc.qt.io/qt-5/qvector.html#resize