Contracts for C++: Revisiting contract check elision and duplication

Timur Doumler (papers@timur.audio)

Document #: P3228R0
Date: 2024-04-16
Project: Programming Language C++
Audience: SG21

Abstract

This paper attempts to inform and structure to the discussion whether the Contracts MVP should retain the rule that checked contract assertions are allowed to be evaluated any number of times when checking a contract predicate, or tighten it to require exactly one evaluation, or possibly some interval of allowed values. We list the different conflicting design requirements and the possible specification solutions, and provide an analysis of which solutions satisfy which design requirements.

1 Introduction

The current Contracts MVP [P2900R6], as forwarded by SG21 to EWG and LEWG for design review, allows contract predicates that have side effects when evaluated. We have consensus that this is useful: when performing a contract check, the user might want to do things like allocating memory (for example, to perform an algorithm that asserts some non-trivial property of a range), lock and unlock a mutex (for example, to assert the value of a variable that can be accessed concurrently), or maintain some state (for example, to assert that the depth of a recursive function call has not exceeded a certain limit). Even [P2680R1], a direction paper proposing that contract predicates with side effects should be ill-formed by default, allowed for "relaxed" predicates that can exhibit such side effects. We therefore need to have a solid specification for how such predicates should behave.

[P2900R6] currently specifies that for a *checked* contract assertion, the predicate evaluation may be elided if the compiler can prove that the result of such an evaluation would be `true` or `false`. Further, it specifies that at any point within a contract assertion sequence, any previously evaluated contract assertions may be evaluated again. Effectively this means that, if a predicate has observable side effects, those side effects may occur zero, one or more times, with no upper bound on the number of possible evaluations.

At the March 2024 WG21 meeting in Tokyo, several issues with this approach were discovered, leading to the realisation that SG21 needs to rediscuss this part of [P2900R6]'s specification and either adopt a different solution that addresses these issues, or at least reconfirm that we consider the current design correct and strengthen the motivation for it. This paper serves to inform and structure that discussion.

2 History and context

In C++2a Contracts [P0542R5], as adopted into the C++20 Working Draft, the issue of contract check elision and duplication did not arise, because evaluating a contract predicate that has observable side effects was specified to be undefined behaviour, and for contract predicates without side effects, elision and duplication is unobservable under the as-if rule.

The paper [P1670R0] proposed to change C++2a Contracts to make predicates with side effects well-defined but allow elision of the predicate evaluation. This paper was never adopted into the C++20 Working Draft because Contracts were removed from it before the paper was considered, however it made its way into an early version of the Contracts MVP [P2388R4], which additionally allowed to *duplicate* the evaluation. This went through several iterations. [P2388R0] allowed to elide or duplicate all (as opposed to "some') side effects of the evaluated predicate, as long as this does not affect the result of that evaluation. [P2388R3] relaxed this to allow eliding or duplicating such side effects per subexpression of a predicate. [P2521R5] strengthened it again to "all side effects".

As work on the Contracts MVP progressed, SG21 spent an extensive amount of time discussing this topic. [P2751R1] proposed to loosen the [P2521R5] model: instead of just allowing elision or duplication, the number of evaluations of a *checked* contract assertion is deliberately made *unspecified*. Such an evaluation can therefore be elided, evaluated once, twice, or even more times, with no specified upper bound. A counter-proposal, [P2756R0], instead proposed to strengthen the [P2521R5] model by specifying that the predicate of a checked contract should be evaluated exactly once. SG21 ended up adopting [P2751R1] and rejecting [P2756R0] (poll results see [P2751R1], Section 5).

Independently from the discussion around these papers, the direction paper [P2680R1] proposed a design direction whereby contract predicates with side effects would be ill-formed by default. However, even this paper provided an escape hatch in the form of so-called "relaxed" predicates that can exhibit side effects, and therefore did not remove the need to specify the behaviour of such predicates (an issue that the paper itself did not address). The design direction proposed by [P2680R1] ultimately failed to get consensus in SG21.

At the March 2024 WG21 meeting in Tokyo, [P2900R6] went through a first round of design review in EWG. This design review has revealed several issues with the current approach:

— A contract assertion that will exhibit undefined behaviour after a number of repeated assertions (say, repeated signed integer addition) can be considered to exhibit undefined behaviour always, as there is no specified upper bound on the number of evaluations;

— Low-latency and real-time systems require a deterministic upper bound on the runtime complexity of a contract assertion;

— For some safety-critical systems, a deterministic upper bound is not sufficient, and a guarantee is required that a checked assertion is evaluated a known, deterministic number of times.

In addition, an EWG guidance poll revealed that a significant number of people prefer that contract assertions should not be allowed to be evaluated more than once (poll results see [D3197R0], Poll 7).

The paper [P3119R0] was written in response to EWG's review. It attempts to address the issues with undefined behaviour and the lack of a deterministic upper bound by introducing an implementation-defined upper bound, and recommending a value of 64. However, the paper does not attempt to address requests by EWG members that the number of evaluations be specified as exactly once or not more than once. Given that the room is currently split on this issue, the solution proposed in [P3119R0] may be insufficient and we may have to at least consider a stronger model if we wish to gain approval by EWG, CWG, and Plenary for adding a Contracts MVP to C++26. The roadmap paper [D3197R0] therefore proposes to rediscuss this issue in SG21.

3 Scope of this paper

This paper does not propose any concrete changes to the Contracts MVP. Instead, it lists the different conflicting design requirements and the possible specification solutions, and provides an analysis of which solutions satisfy which design requirements. The intent of the paper is to add some structure to the discussion and to highlight the engineering tradeoffs that each solution involves, to help SG21 reach consensus.

We do not consider *unchecked* contract assertions (those having the *ignore* evaluation semantic), as such evaluations do not involve evaluating the predicate.

Further, we do not consider contract predicates that have no observable side effects. Under the as-if rule, it is unobservable whether such "pure" predicates are evaluated zero, one, or multiple times, as long as the compiler has correctly determined whether the result of such an evaluation would be `true` or `false`. We therefore only consider contract predicates that have *observable side effects* when evaluated.

Finally, we do not consider contract assertions whose evaluation may end up being elided because of undefined behaviour occurring either during evaluation of the predicate itself or elsewhere in the program. We only consider programs that would have well-defined behaviour when the predicate is evaluated once.

4 Design requirements

In this section, we summarise the different, partially conflicting design requirements that have motivated different proposals in this space.

4.1 Deterministic number of evaluations

There are several reasons why a guarantee that the predicate is evaluated a deterministic number of times may be desirable.

More generally, a deterministic number of evaluations is required if we wish to minimise the amount of implementation-defined and unspecified behaviour added to the C++ language by the Contracts MVP. This is something that has been brought up as a significant concern in EWG.

More specifically, deterministic behaviour is a requirement in some safety-critical systems, which may be unable to use Contracts at all unless this requirement is satisfied. Note that this does not necessarily mean a deterministic execution time, or a deterministic sequence of CPU instructions, but a deterministic observable behaviour with regard to side effects. In other words, eliding the evaluation of a side-effect-free predicate would be a normal optimisation transformation and unobservable under the as-if rule, while eliding the evaluation of a predicate with side effects would make the behaviour of a *checked* contract assertion observably non-deterministic, which is undesirable. The only situation where this happens in the language today is copy elision; for some use cases, it may be considered undesirable to add more such situations.

4.2 Exactly one evaluation

If we constrain "deterministic number of evaluations" further to "exactly one evaluation", we gain more properties that one might consider desirable. "Exactly one evaluation" seems to be the only reasonable implementation of "deterministic number of evaluations" (we see no possible reason why one would want to specify, for example, that each contract assertion is always evaluated twice), therefore we consider them to be the same requirement for the remainder of this paper.

[P2756R0] argues that "exactly one evaluation" is the most simple, intuitive, and easy to reason about solution, and the only one that follows existing practice in `assert` and similar macros —

meaning that this is the behaviour a user would expect when upgrading their uses of such macros to [P2900R6] contract assertions.

Assertions that rely on being evaluated exactly once when checking is enabled are common in practice. For example, one might want to add an assertion counting the number of times a recursion occurs, and report a contract violation if the recursion depth exceeds some fixed limit. Consider the following example (adapted from Clang):

```
#ifndef NDEBUG
  unsigned depth = 0;
#endif
  while (item->isVariant()) {
    assert(++depth < 6 && "Variants are nested deeper than the supported limit");
    // ...
  }
```

Without a guarantee that the predicate is evaluated exactly once when the assertion is checked, this code would break, as the assertion could end up failing when it should pass, or passing when it should fail, potentially leading to undefined behaviour.

The assertion as written above would not work with the current Contracts MVP because [P2900R6] does not offer a facility like `NDEBUG` for conditionally declaring a variable only when the associated contract assertion is checked. Further, in [P2900R6], local variables such as `depth` are implicitly const and would need to be wrapped into a `const_cast` in order to be incremented in the contract predicate. However, even without any of these limitations, any predicate that maintains and updates state like the one above cannot ever be made to work if there is no guarantee that *checked* contract predicates are evaluated exactly once.

Such predicates exist and are relatively common in existing code bases, as can be easily verified by a code search[1]. Further, it is possible that there are cases where such a predicate, when used with `assert`, does not wrap the declaration of the counter in an `#ifdef NDEBUG` block, and the counter is not a local variable, but one with static or thread-local storage duration, meaning that changing such an `assert` to a `contract_assert` would silently break the program and potentially introduce undefined behaviour, without any compiler warning or other message to the user.

[P2900R6] argues that such predicates should not exist, as they modify the state of the program they are supposed to merely observe, violating the design principles that the Contracts MVP is based on. We can also argue that the Contracts MVP is deliberately not designed to serve as a drop-in replacement for `assert` and should not be used in that way. This is all well and good, but in practice users will likely use the Contracts MVP in that way anyway: they are used to "exactly one evaluation when checked" from existing assertion macros and will intuitively expect the same behaviour. The possibility of silent code breakage and introduction of undefined behaviour by turning deterministic into non-deterministic evaluation is arguably a particularly user-hostile way to break these users' assumptions, and could significantly hamper the adoption of Contracts.

4.3 Deterministic upper bound on number of evaluations

Many low-latency and real-time systems do not necessarily require full deterministic behaviour, but do require a deterministic upper bound on the runtime complexity of a contract assertion. These systems may be unable to use Contracts at all unless this requirement is satisfied.

In addition, without such a deterministic upper bound, a contract assertion that will exhibit undefined behaviour after a number of repeated assertions (say, repeated, accumulating signed integer addition) can be considered to exhibit undefined behaviour always; a particularly hostile compiler may treat such contract assertions as unreachable code while being conforming. Both of these issues are discussed in more detail in [P3119R0].

[1] https://codesearch.isocpp.org/cgi-bin/cgi_ppsearch?q=assert%28%2B%2B&search=Search

There are two ways to specify such an upper bound: either normatively specify a concrete number in the Standard (for example, "at most two evaluations"), or merely specify that an implementation has to define *some* deterministic upper bound but leave the actual number unspecified. The latter is proposed by [P3119R0]. For consumer-facing real-time applications like video games and audio processing software which often need to support different compilers, a normatively specified upper bound seems preferable to an implementation-defined one, because the latter could change across compilers or even across different versions of the same compiler, making it harder to reason about the code and the guarantees it provides. In addition, it is unclear to what degree either option is acceptable instead of the stricter "deterministic number of evaluations" guarantee for such applications; this will most likely depend on the concrete use case.

4.4 Allow duplications

The main motivation for allowing duplication of predicate evaluations is to allow the implementation to perform both caller- and callee-side checking while preserving ABI compatibility.

Note that only a subset of contract checks can be implemented caller-side[2]. Some contract checks must be performed callee-side, for example, when a function is called indirectly, such as through a function pointer or a facility like `std::function`[3], or when checking postconditions in the Microsoft ABI, as this ABI performs argument destruction callee-side and postcondition checks are guaranteed to happen before argument destruction.

Now, consider a shared library that has function contract assertions on its function declarations, and an application that uses this library by compiling against a static library header and then dynamically loading a compiled library binary. The developer of the application needs to have the choice of compiling the application with caller-checks either disabled or enabled (the latter resulting in the caller-checkable subset of the library checks being checked, which can be useful to diagnose problems). Likewise, the provider of the shared library needs the choice to ship the compiled library binary with callee-side checks disabled or enabled (the latter being a "hardened" version of the library). Either version of the application should be able to use either version of the library binary without having to recompile and re-link[4]. The combination of the application having caller-side checks enabled and the library having callee-side checks enabled will result in some library checks being checked twice.

We can consider implementation strategies that would allow this use case while also guaranteeing that contract checks be evaluated exactly once (or not more than once), but such strategies come with tradeoffs. One possible strategy would be to compile the library binary such that each function with precondition or postcondition assertions has two entry points, one that performs the callee-side check and one that does not. However, this can significantly increase the amount of symbols in the binary, and requires a change to the ABI. Another possibility is to compile the library such that the choice whether to perform callee-side checks is made dynamically, but this incurs additional runtime overhead and again requires a change to the ABI.

[2]Note that, if we adopt the proposed design in [P3097R0] and [P3165R0] for supporting function contract assertions on virtual functions, the reverse will also become true: only a subset of contract checks can be implemented callee-side. In particular, checks for function contract assertions of the statically called function in a virtual function call can only be generated caller-side as the statically called function is unknown to the callee.

[3]In such scenarios, a callee-side check can be generated only if the compiler front-end sees the function contract assertion related to the function call, i.e. it knows which function will be called and can see the declaration of that function.

[4]Without this requirement, guaranteeing that contract checks be evaluated exactly once would be relatively easy: If in the library, checks are disabled, in the application we call a thunk wrapper, which checks precondition assertions and then calls the function; otherwise, we just call the function. The library's build never affects whether the preconditions are checked, only the application build does. However, in this implementation model, we cannot flip precondition checks on and off by just building the library differently and then running an unmodified application with it; instead, we would need to recompile the application, which might be prohibitively slow or impossible.

For some users, the tradeoffs of either strategy will be unacceptable: a global ABI break might make this approach undeployable, and in a world where some C++ applications require tens of gigabytes of memory to link due to the sheer amount of symbols, there is a strong incentive to avoid adding more symbols. Further, there is currently no implementation experience for either strategy[5]. Avoiding such tradeoffs however requires us to either explicitly allow duplication of predicate evaluations, or to place this use case outside of the scope of the C++ Standard and treat support for it as a non-conforming vendor extension, as we do for example for `-fno-exceptions`.

[P2751R1] discusses other reasons why allowing duplication would be beneficial.

4.5 Allow elisions

One might want to allow eliding contract predicates along with their observable side effects, in a fashion similar to copy elision, if the compiler can prove that the result of the elided evaluation would be `true` or `false`. For example, if a function `f` has a checked precondition assertion `pre(x)`, and immediately after that, as the first statement in the body of `f`, another function `g` with a functionally equivalent precondition assertion is called, eliding the latter assertion would avoid unnecessarily evaluating the expression `x` twice. This is discussed in more detail in [P2751R1].

4.6 Allow more than two repetitions

While supporting both caller- and callee-side checking in the same program without an ABI break or extra symbols may require allowing evaluation to occur twice, we are not aware of any scenario where evaluation could end up occurring three times or more. It seems therefore that it would be enough to allow evaluation to occur up to twice, rather than an unspecified number of times. However, [P3119R0] describes a use case for which allowing an unspecified number larger than 2 might be useful. A user might want to test for destructive contract assertions (contract assertions with undesirable side effects) by evaluating them repeatedly during testing and observing if results change. Such tests are a good way to gain confidence that the contract predicates are not doing something that fundamentally breaks the program's behaviour: destructive effects are more likely to be noticeable when evaluating the predicate, say, 16 times, than when evaluating it twice. By allowing more than two repetitions, the specification would allow a compiler to provide a conforming option to request an arbitrary number of repetitions for performing such tests.

4.7 Make the number of evaluations unspecified

One might want to make the number of evaluations deliberately unspecified to further discourage users from writing predicates with side effects as they learn that they cannot rely on the behaviour of such predicates: the side effects may occur any number of times or not at all. This is discussed in more detail in [P1670R0] and [P2751R1].

Not relying on side effects in contract predicates is stated as an important design principle in [P2900R6]. The philosophy is that contract assertions should be able to test the correctness of the programs that did not yet have those assertions in them, which effectively means that they need to test the correctness of the same program when those assertions are ignored and not some other program. If, however, the predicates alter the behaviour such that the new, altered program is then correct (or incorrect), such contract assertions fail at diagnosing any defects and instead themselves introduce Heisenbugs. Note that assertions with side effects are nevertheless found in real-world code bases, and can be useful, such as the the recursion depth counter described in Section 4.1.

[5] Caller-side checking itself does not have implementation experience either, but it is arguably somewhat less theoretical, because compilers know how to parse the function contract specifiers of a function declaration and rewrite a function call `f()` into a caller-side checked function call such as `(pre_check() ? f() : abort())`.

4.8 Avoid numbers other than 0, 1, "many"

Another possible reason to prefer an unspecified number of evaluations instead of "up to 2" is the wish to avoid magic numbers like 2 in the specification of the C++ Standard, and stick to the options 0, 1, or "many" that are preferred in some computer science contexts.

5 The solution space

We begin by listing all plausible specification strategies for how many times the predicate of a checked contract assertion may be evaluated. Whether elisions are allowed is orthogonal to the other concerns, so we can split the solution space into solutions that allow elisions and solutions that do not. The four known solutions that allow elisions are as follows (all four have been proposed somewhere at some point):

A_0. At most once, i.e. evaluation may be elided but not duplicated ([P1670R0], [D3197R0] poll 7).

B_0. At most twice ([P2521R5]), i.e. evaluation may be both elided and duplicated.

C_0. An unspecified number of times, with an implementation-defined upper bound N ([P3119R0] proposal 3).

D_0. An unspecified number of times, with no upper bound (status quo, [P2900R6]).

Further, we can construct four solutions that are analogous to the above but do not allow elisions (only one of those, A_1, has been formally proposed):

A_1. Exactly once ([P2756R0]).

B_1. Once or twice, i.e. evaluation may be duplicated but not elided.

C_1. An unspecified number of times, with an implementation-defined upper bound N, but at least once.

D_1. An unspecified number of times, with no upper bound, but at least once.

Remember that we only consider evaluations of contract assertions with *checked* evaluation semantics (`observe`, `enforce`, or `quick_enforce`), as evaluations with *unchecked* evaluation semantics (`ignore`) always evaluate the predicate zero times and we do not consider removing `ignore` from the MVP or changing how it is specified. Remember further that we only consider evaluation of predicates with observable side effects. For predicates with no observable side effects, neither elision nor duplication are observable under the as-if rule, and therefore all of the above solutions are equivalent.

A few more solutions than the ones listed above are theoretically possible, such as requiring a deterministic number of evaluations that is not once (e.g. "every contract assertion must always be evaluated twice"), or requiring a normatively specified (as opposed to implementation-defined) upper bound larger than two. We do not consider these solutions here because we are not aware of any benefits these might have over the ones listed above.

Further, there has been a suggestion that we could allow duplications and simultaneously ensure a deterministic amount of evaluations with a solution that says that the predicate must be evaluated exactly N times, where N is implementation-defined. An implementation may then say that, for example, N is 1 if the function called is in a statically linked library, but 2 if the library is linked dynamically. However, a conforming implementation may satisfy such a specification by simply saying that N is any number between 0 and 64 (for example), therefore such a solution is equivalent to solution C_0.

Design requirement Nr. of evaluations:	A_0 $0-1$	A_1 1	B_0 $0-2$	B_1 $1-2$	C_0 $0-N$	C_1 $1-N$	D_0 $0-\infty$	D_1 $1-\infty$
1. Normatively specify a deterministic number of evaluations / exactly one evaluation	no	yes	no	no	no	no	no	no
2. Normatively specify a concrete, deterministic upper bound on number of evaluations	yes	yes	yes	yes	no	no	no	no
3. Require *some* deterministic upper bound on number of evaluations	yes	yes	yes	yes	yes	yes	no	no
4. Allow duplication of evaluations to facilitate caller-/callee-side checking compatibility	no	no	yes	yes	yes	yes	yes	yes
5. Allow elision of evaluations if compiler can prove predicate would evaluate to `true` or `false`	yes	no	yes	no	yes	no	yes	no
6. Allow more than two repetitions to test for destructive contract assertions	no	no	no	no	yes	yes	yes	yes
7. Make number of evaluations unspecified to discourage predicates with side effects	yes	no	yes	yes	yes	yes	yes	yes
8. Avoid specifying numbers of evaluations other than 0, 1, "many"	yes	yes	no[1]	no[1]	yes	yes	yes	yes

Table 1: Decision matrix for the number of evaluations problem for `pre` and `post`.

6 Choosing the best solution

6.1 `pre` and `post`

Now that we enumerated both the requirements and the possible solutions, we can create a decision matrix listing which possible solutions satisfy which design requirements. The decision matrix for `pre` and `post` is given in Table 1.

The design requirements listed in the decision matrix are numbered to make it easier to refer to them. These numbers do not imply a ranking by importance; we are not attempting to perform such a ranking in this paper.

This decision matrix reveals that solution A_1 (predicate is evaluated exactly once) is the only solution that satisfies the design requirement of having a deterministic number of evaluations, while solution C_0 (predicate is evaluated any number of times, with an implementation-defined upper bound) satisfies all other design requirements except the deterministic number of evaluations, and all other solutions satisfy some subset of the requirements satisfied by C_0. Satisfying all design requirements in this set simultaneously is therefore impossible.

Choosing a solution will require making decisions about contradicting requirements. For example, SG21 will need to decide whether it is more important to have a normatively specified deterministic number of evaluations (requirement 1), or whether it is more important to allow duplications of

[1]Note that the 0, 1, "many" requirement might be satisfiable for B_0 and B_1 if instead of generally allowing 2 evaluations, we specified these cases as $0-1$ evaluations on one or both sides of the call (caller-side and callee-side).

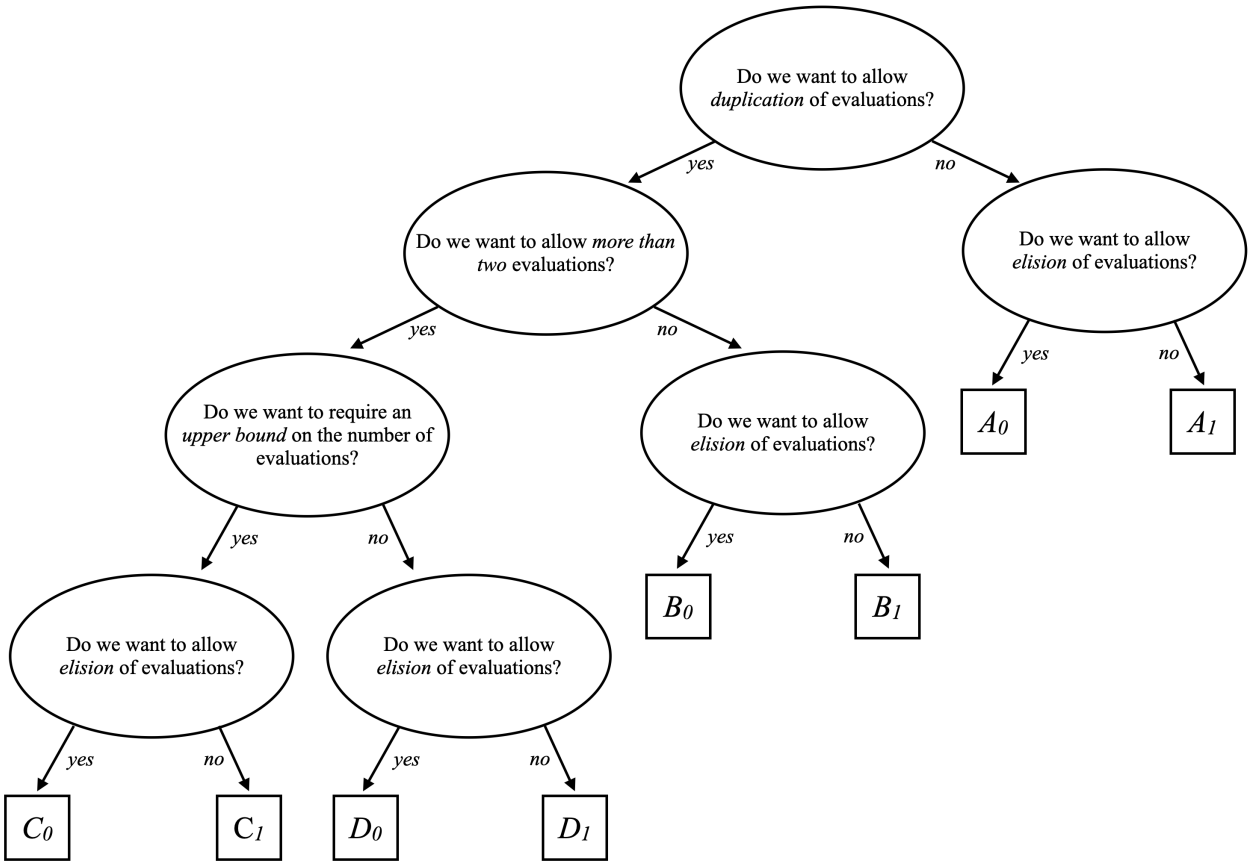

Figure 1: Possible decision tree for picking a solution.

evaluations (requirement 3) in order to support the caller-side vs. callee-side use case described in Section 4.4, as satisfying both of these requirements is impossible[6]. Other decisions involve whether to allow elision and whether to allow an unspecified number of evaluations.

A possible decision tree to pick a concrete solution for `pre` and `post` is illustrated in Figure 1.

6.2 `contract_assert`

For `contract_assert`, the decision matrix looks slightly differently. The need to support caller-/callee-side checking compatibility, which is the motivation for requirement 4, does not exist for `contract_assert`, because assertion statements can only appear inside the function body and are therefore always checked callee-side. Instead, there is a new requirement: it might be desirable to pick the same solution for `contract_assert` as we did for `pre` and `post` so that the three assertion kinds provided in the Contracts MVP behave in a consistent fashion.

Violating this consistency requirement would have implications for teachability, complexity of the language, etc. It would also widen the gap between the use of `pre` and the use of `contract_assert` at the start of a function, or the use of `post` and the use of `contract_assert` before returning, which is the only mechanism offered by [P2900R6] to insulate precondition and postcondition checks from client translation units when the developer considers them an implementation detail.

That said, the consistency requirement should be be weighed against the other requirements and we can in principle adopt a different solution for `contract_assert` than for `pre` and `post`. For

[6] Polling this particular decision would largely be a re-litigation of the decision made at the February 2023 WG21 meeting in Issaquah when SG21 polled [P2756R0] and [P2751R1], respectively.

example, if we decide that requirement 1 (deterministic number of evaluations) is particularly important for `contract_assert`, because of the issue regarding existing practice with the `assert` macro discussed in Section 4.1, and that this is more important than the consistency requirement, or the requirements 5 — 7 when applied to `contract_assert`, we could adopt solution A_1 for `contract_assert` despite having chosen a different solution for `pre` and `post`.

Acknowledgements

Thanks to John Spicer, Ville Voutilainen, and Gašper Ažman for the illuminating discussions that led to this paper. Thanks to Jonas Persson, Andrew Tomazos, Joshua Berne, and Jens Maurer for their helpful comments on a draft of this paper.

References

[D3197R0] Timur Doumler and John Spicer. A response to the Tokyo EWG polls on the Contracts MVP (P2900R6). https://wg21.link/d3197r0, 2024-04-04.

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. https://wg21.link/p0542r5, 2018-06-08.

[P1670R0] Alisdair Meredith and Joshua Berne. Side Effects of Checked Contracts and Predicate Elision. https://wg21.link/p1670r0, 2019-06-06.

[P2388R0] Andrzej Krzemieński and Gašper Ažman. Abort-only contract support. https://wg21.link/p2388r0, 2021-06-15.

[P2388R3] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. https://wg21.link/p2388r3, 2021-10-13.

[P2388R4] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. https://wg21.link/p2388r3, 2021-11-15.

[P2521R5] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum. Contract support – Record of SG21 consensus. https://wg21.link/p2521r5, 2023-08-15.

[P2680R1] Gabriel Dos Reis. Contracts for C++: Prioritizing Safety. https://wg21.link/p2680r1, 2022-12-15.

[P2751R1] Joshua Berne. Evaluation of *Checked* Contract-Checking Annotations. https://wg21.link/p2751r1, 2023-02-14.

[P2756R0] Andrew Tomazos. Proposal of Simple Contract Side Effect Semantics. https://wg21.link/p2756r0, 2022-12-31.

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r6, 2024-02-29.

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman. Contracts for C++: Support for Virtual Functions. https://wg21.link/p3097r0, 2024-04-15.

[P3119R0] Joshua Berne. Tokyo Technical Fixes to Contracts. https://wg21.link/p3119r0, 2024-04-03.

[P3165R0] Ville Voutilainen. Contracts on virtual functions for the Contracts MVP . https://wg21.link/p3165r0, 2024-02-16.