Configuration of Contract Evaluation Semantics

Document #:	P3595R0
Date:	2026-07-15
Project:	Programming Language C++
Audience:	SG15 (Tooling)

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Iain Sandoe <iain@sandoe.co.uk>

Abstract

This paper is not a proposal for the C++ Standard, but rather a description of how compilers could choose to expose a robust configuration system for managing the behavior of contract assertions. Prototype implementations in both GCC and Clang have been made available.

Contents

Revision History

Revision 0

- Original version of the paper

1 Introduction

Contract assertions in C++ can be evaluated with one of four different semantics, *ignore*, *observe*, *enforce*, and *quick-enforce*. By design, each evaluation of a contract assertion can be made with a different semantic from any previous evaluation of the same assertion (either within the same TU, or between TUs). Currently, experimental implementations of C++26 Contracts provide much simpler controls over the semantic (which do not implement the fine-grained selection but, rather, express semantics at the TU, or higher, level). While such simpler controls are conformant, it is desirable that we specify common ways in which the long-term objective of detailed description can be met.

To augment the C++26 Contracts baseline, mechanisms have been proposed that allow code authors to specify the finer granularity in the source code [P3400R4]. This paper proposes an approach for improved build-time facilities that give users control over contract assertion behavior, both independently of and in conjunction with the source-code tools the language itself would provide.

The build-time controls serve the needs of the consumer of code that has contract annotations; these needs can be disparate from those of (and/or unknowable by) the code-author and might include:

- Sometimes, an assertion might trigger a compiler bug with some semantics — providing a workaround to that sort of problem is essential.
- Sometimes an assertion might be hit and it might be determined that the assertion itself is incorrect, and disabling it externally to the source code is the only available option. In cases such as this, the `source_location` on the `contract_violation` object might be the only information available that identifies the contract assertion.
- Just like profile-guided optimization, the ability for static analysis tools to produce a report of contract assertions which are provably true and can be *ignored* might be a powerful tool for improving the viability of running checked builds without unreasonable run-time costs.
- Until we get labels to allow per-contract control [P3400R4], this facility provides a mechanism to do things such as *observe* a newly added contract assertion while leaving the semantic choice for all other assertions unchanged.

The specification of fine-grained control over (potentially individual) contract assertions is not thought to be practicable using command-line controls and will, instead, be better accomplished using a configuration file.

To that end, we want to propose a file format that can be used for the specification of contract assertion evaluation semantics. This format aims to meet a number of basic requirements:

- It needs to be extensible so that new facets of control are easily added. We do not want to require perfect parity in feature support between all compilers because that will limit progress.

- We need a well-understood response to what to do when encountering a piece of configuration that is not understood by the current compiler.
- The same configuration should be consumable by every TU in a program so that we get predictable semantics for each contract assertion if all compilers involved support the feature set being used.
- How the new configuration interacts with other configuration options must be well defined. For this, we recommend that other configuration options that impact the evaluation semantic all be rephrased in terms of adding additional configuration files with specific contents, along with a well-defined order in which those configurations are applied.
- Applying the configuration to determine the evaluation semantic must be an operation that can be reasonably optimized. Milliseconds in this calculation can add up quickly — although it need not aim to be significantly faster than the cost of generating the code for the contract assertion itself. Overall, we must specify an algorithm that will remain reasonably bounded.

Then we should consider the different aspects we want to, at a minimum, support configuring, i.e. the primary use cases we are targeting:

- It should meet the top-level requirements described above.
- Per-TU evaluation semantics should not go away with this adoption.
- It might be of interest for some use cases to base enablement on the contents of the assertion predicate.
- We want to be able to identify contract assertions by the component to which they belong. All contract assertions belong to a particular function, but for a function, *component* can have a number of meanings that might be the right choice for different users:
  - The namespace of the function.
  - The module to which the function is attached.
  - The file name where the function is declared. Note that file names can in general be challenging to use and resolve, as compilers must often equate situations where a file is found and known through one name while a user identifies through a completely different one — whether because of paths compounded in different ways, symbolic links, or varying filesystems.
  - The translation unit compiling the file. Note that this also confuses many users because, for inline functions, it is not always obvious which translation unit's version of an inline function will end up used for any given invocation of that function.
- For caller-side checks, and possibly even for checks within a function that has been inlined into an enclosing caller, we might similarly want to base our semantic off of the component of the caller and not the component to which the contract assertion directly belongs.
- When both the calling and called component are known and differ, it can be good to base enforcement on how much we trust the two components. When they are the same, we might apply a different set of logic.

2 Proposal

For every contract assertion that is encountered, we will need to determine how it should be expanded (i.e. what its active semantic is). The more information we can pin down at compile time about the chosen semantic, the better the code we can generate; however there remain use cases for deferring the final selection of semantics until link-time or even run-time.

Contract assertions will potentially require code generation in two positions:

1. When functions with assertions (`pre`, `post`, `contract_assert`) are defined.
2. When caller-side checking is enabled, for any called function that declares `pre` or `post` conditions.

For 1, there is not too much concern. This lookup is an action carried out once per TU for each assertion.

For 2, we need to make careful provision; the processing that is implied could be required at every call-site, and for each `pre` and `post` condition of the callee. We cannot, in general, rely on caching decisions since the intent is to allow these very decisions to vary between instances.

The strategy suggested here is that the semantic selection algorithm be bounded and defined ahead of time; this means that it can be fully optimised as an integral part of the compiler.

The selection of active semantic must then be specified by control values to this fixed algorithm.

In order to guide the specification of the selection algorithm we will start by examining the kinds of control value that might be used.

2.1 Input Values

The inputs that are available when doing the code generation for a contract assertion include a variety of properties, and depending on which part of code generation is considering whether to emit code for the contract assertion:

- The `kind` of contract assertion — `pre`, `post`, or `contract_assert`, which will not vary.
- The contents of the expression in the predicate. It might be interesting to, for example, enable all contract assertions that match a regular expression `.*` = nullptr!, but the cost of applying regular expressions or even exact string matching to what will be an AST of tokens when contract assertion code is being generated is prohibitive and will be finicky.
- The location of the contract assertion as filename and line number (matching what would be produced by `std::source_location::current()` or `__FILE__` and `__LINE__`). This is, of course, not a very stable value as it can vary based on whether a function has a redeclaration, or between the canonical declaration in a well-maintained header and the function's definition in an implementation file. Even so, the ability to pinpoint a specific assertion for enablement/disablement can open up significant opportunities for external tooling.
- For any contract assertion, information about the function to which the contract belongs:
  - Its namespace, which could be explicitly matched, or matched by prefix. We would also need to decide how to treat inline namespaces, especially nested inline namespaces if

doing prefix matching.

  - The module to which it is attached, if any. Similarly to namespaces, we could match exact modules, or we could match all modules with certain prefixes (and the common `'.'` delimiter between identifiers).

  - The name of the function. This can, of course, be challenging to compare and resolve because there are a wide variety of ways to spell the name of the same function, and to be useful any matching algorithm would have to reconcile those differences.

- For a caller-side evaluation of a precondition or postcondition assertion, the same information about the function to which the invoking function belongs.

  - For inlined functions, one could also consider the configuration that would apply to caller-side precondition and postcondition assertions when deciding how to generate code for assertion statements (`contract_assert`) in the function body being inlined.

- The Clang `[[clang::contract_group(...)]]` attribute, or group labels as specified in [P3400R4], both specify a group (that is a string) that can be attached to a contract assertion.

- Whether the contract assertion is being evaluated at runtime or during constant evaluation.

2.2 Configuration Outputs

The simplest case for outputs is to have this configuration produce one of the known contract evaluation semantics — *ignore*, *observe*, *enforce*, or *quick-enforce*.

Compilers may also choose to support additional implementation-defined semantics, in which case they should document what values they support and what that semantic will mean.

If a contract assertion's semantic should be determined at link time, then that should be specified along with a default semantic and a link-time variable to reference to determine the chosen semantic. Such link-time configuration of contract assertions will likely be conditionally supported.

A function invocation could equally be used instead of a variable, which could be passed any of the known information about a contract assertion (such as its kind, location, groups, etc.) in order to make more nuanced link-time decisions. Link-time optimization when applied to such a function could turn any function that has a deterministic output into final code that has the same properties as if the semantic had been chosen at compile time.

Support for a runtime query to determine evaluation could also be conditionally supported by specifying a function name to use that would be expected to return a semantic. Naming a function with no arguments that returns a `std::contracts::evaluation_semantic` is probably required, as anything more involved would need to also have the complexity of pulling in a declaration for whatever is being used somehow.

2.3 Matching Criteria

For some inputs, we want to provide filtering data that matches exactly the values on the contract assertion. For others, we want to define a more flexible choice that allows control at different granularities.

- For location information we want to be able to specify either a file, a file and a range of line numbers, or possibly even a file, range of line numbers, and a column number for the rare occasions where such precision is needed. (Note that location information with files and line numbers can be very brittle to use in practice, but is also a key way in which static analysis will convey guidance on semantics to detached stages of a toolchain during a larger build process.)
- Modules, namespaces, and contract groups might all benefit from specifying a prefix and not just an exact match. For namespaces in particular this can be important to avoid having to distinguish inline namespaces that a user might be unaware of. We should be specific about the delimitation we use for such matching – for namespaces `::` is natural, while for modules and contract groups a `.` seems to be a more common convention.

2.4 File Format

After a brief discussion with two major compiler vendors, JSON seemed to be indicated as the easiest general structure to specify our configuration in. We want something easily extensible, fully specified, and where writing parsers for the compilers will not be a large investment in its own right. Implementation of this strategy was fairly straightforward in both Clang and GCC.

Any entry that specifies an input or output type that is not supported should be skipped, although compilers might wish to provide an optional warning to indicate that this is happening. Possibly, producing such a warning during parsing of the configuration files the first time an input or output type that is not supported is encountered might be viable, but it should be remembered that these configuration files will be most successful if they can be passed with confidence to all compilers building a program.

To that end, we'll start with a general JSON format that separates the matching criteria from the output configuration, making it very clear when a compiler might be encountering an entry that is not supported.[1]

```
{
  "type": "object",
  "properties": {
    "match": {
       ...
    },
    "output": {
       ...
    },
  },
  "required": [ "output" ]
}
```

The `match` property will be used to capture different criteria that will be compared to a contract assertion's values to determine if this is the output to use. When processing the configuration, it will always be done as-if linearly and the first entry that matches will use its output to determine how code is generated for that particular contract assertion.

[1] We'll specify these entries using a simple JSON schema, see https://json-schema.org/ for more information.

```
...
    "match": {
        "kind" : "pre" | "post" | "contract_assert",
        "group" : "string",
        "location" : "filename"
                   | "filename: list-of-intervals"
                   | "filename: line: column",
        "namespace" : "string",
        "module" : "string",
        "function" : "function declaration",
        "constexpr" : "boolean",
        "caller" : "boolean"
                 | {
                     "location" : "filename"
                                | "filename: list-of-intervals"
                                | "filename: line: column",
                     "namespace" : "string",
                     "module" : "string",
                     "function" : "function declaration",
                   }
    },
...
```

All of these properties are optional. When matching a caller-side check that is redundant to an expected callee-side check — i.e., a caller-side check that is not caller-facing[2] — the caller property must be there and its contents must match the caller location information, or it must be the boolean value `true`.

Whenever there is a match, the corresponding output determines how code will be generated for the contract assertion evaluation:

```
...
    "output": {
        "semantic" : "ignore"
                   | "observe"
                   | "enforce"
                   | "quick-enforce",
        "dynamic" : {
            "linkage" : "C" | "C++",
            "name" : "string",
            "provideweak" : "boolean"
        }
    }
...
```

If `dynamic` is specified, the semantic will be determined by invoking a function with the linkage and name (with no arguments, that must return a `std::contracts::evaluation_semantic`) specified

[2]When contract support on virtual functions is reintroduced it is expected that we will have caller-facing contract assertions that the callee is not going to also evaluate, and in those cases we do not want to treat their evaluation as opt-in.

in the value. When `provideweak` is not specified (or `true`) a weak version of the function will be emitted that returns the `semantic` specified for the output.

- Dynamic selection allows distributing binaries that have a preferred semantic but do not require rebuilding to use a different semantic.
- Different functions can easily be assigned to different libraries, contracts with different groups, or contracts with different categories.
- When a simple function that just returns a semantic is linked in, link-time optimization will be able to recover some or all of the codegen quality that a fully compile-time chosen semantic would have achieved.
- If there is a [P3400R4] allowed-semantics label, the returned semantic will be adjusted to be one of the allowed semantics.
- If there is a [P3400R4] compute-semantics label, the returned semantic will be passed through the `compute_semantic` function and the result of that function will be used. (An enforced violation will occur if the resulting semantic is not an allowed semantic).
- A dynamic function cannot be used at compile time, and `semantic` will be used instead (if specified). If the `semantic` key is not specified this match will be treated as if `constexpr` was `false` (and so the match will apply only to runtime evaluations).

For example, a configuration might select the semantic for a library's contracts through a function provided by that library:

```
{"match": {"group": "mylib"},
 "output": {"semantic": "enforce",
            "dynamic": {"linkage": "C++", "name": "mylib::contract_semantic"}}}
```

We recommend giving such a function a library- or component-scoped name so that independent libraries or groups can each supply their own selection function without collision. With `"C++"` linkage the name may be fully qualified and is mangled as usual; with `"C"` linkage the name is used verbatim as the symbol, which allows targeting a specific (including externally mangled) function when needed.

The selection function returns a `std::contracts::evaluation_semantic` by value, so its calling convention depends on that enumeration's underlying type. For a single selection function to be shared across a program — and especially across toolchains through `"C"` linkage, where one toolchain may produce the symbol another consumes — the implementations must agree on that underlying type. Our two prototype implementations initially disagreed (one used a 16-bit underlying type, the other an 8-bit one), which would have made a shared selection function ABI-incompatible across toolchains; we have reconciled both on a 16-bit (`uint16_t`) underlying type for `evaluation_semantic` so that a single selection function is safely shareable. We therefore recommend that the Standard fix the underlying type of `evaluation_semantic`.

2.5 Interaction With Existing Configuration Flags

When no contract configuration is provided, the compiler should specify an exhaustive set of entries that are always last in the list of entries that are checked. We recommend the following to achieve a

default of *enforce* for all checks as recommended by the Standard, with no double-checking caller-side without explicitly requesting it:

```
{"match": {"caller": true}, "output": {"semantic": "ignore"}}
{"output": {"semantic": "enforce"}}
```

As part of a compiler configuration a series of contracts configuration files will be determined by compiler flags. These will always be checked, in order, first:

```
gcc -fcontract-configuration-file=myfile1.json,myfile2.json
```

After checking all of the configuration files, additional entries will be checked based on other compilers flags, where certain compiler flags are treated as equivalent to corresponding JSON specifications being appended to the end of the list of entries to process:

Flag	JSON
`-fcontract-evaluation-semantic=<semantic>`	`{"output": {"semantic": "<semantic>"}}`

The ongoing GCC implementation of contracts has, for example, added a flag `-fcontracts-client-check`, that when passed on the command line takes 3 possible arguments, `none`, `pre`, and `all`, that map to different JSON entries that would be added in addition to the entry added by simply specifying `contract-evaluation-semantic`:

Flag	JSON
`-fcontract-evaluation-semantic=<semantic>` `-fcontracts-client-check=all`	`{"match": {"caller": true},` ` "output": {"semantic": "<semantic>"}}`
`-fcontract-evaluation-semantic=<semantic>` `-fcontracts-client-check=pre`	`{"match": {"caller":true, "kind": "pre"},` ` "output": {"semantic": "<semantic>"}}`

Standard Library hardening, currently enabled with the `-fhardened` flag passed to Clang, can be implemented in the same way:

Flag	JSON
`-fhardened`	`{"match": {"namespace": "std", "group": "hardening"},` ` "output": {"semantic": "enforce"}}`

Other compiler flags can easily be added in addition to these.

3 Optimization Opportunities

The question comes up about how expensive the specification defined here will be to evaluate when it must be processed for every single contract assertion.

We suggest a few key optimizations that will make it palatable to implement and deploy:

- The namespace, module, file information, function name, and any other information should be passed to the function that processes the entries in a form that allows for lazy evaluation for the typical case where that information will never be needed by the configuration, but also for the case where it might be needed and used multiple times for different contract assertions that are at the same point of evaluation.
- It is likely that systems will be built that generate comparatively large configurations that control the semantics of contract assertions at a very granular (and thus verbose) level. These configuration files will consist of many entries that differ by specifying non-overlapping file and line number ranges. Such consecutive entries should be collapsed into a simple data structure that enables a fast map lookup and interval check to identify the configured semantic for a particular file and line number.

4 Implementation Experience

This configuration system has been partially implemented in branches of GCC and Clang that are available on compiler explorer.

- Control of evaluation semantic based on location: https://godbolt.org/z/e67M7s43e
- Control of evaluation semantic based on group labels (as provided by [P3400R4]): https://godbolt.org/z/Pq6hEv34T
- Control of evaluation semantic differing between compile time and runtime: https://godbolt.org/z/K9nW1eGYK
- Dynamic selection of the evaluation semantic at link or run time via a named function: https://godbolt.org/z/3MMxarYnd

Both compilers support the following options:

- Loading a JSON file using the command line argument `-fcontract-configuration-file=<filename>`.
- Passing JSON entirely on the command line using `-fcontract-configuration=<json>`
- Matching based on location, namespace, kind, group name, and whether evaluation is `constexpr`.

GCC, which is currently the only implementation that supports caller-side checking, also supports enabling caller-side checking through configuration files. This includes being able to control caller-side checking based on caller location or namespace. Examples of this behavior can be found here: https://godbolt.org/z/n94Yob3c3.

Future work remaining to be prototyped in both compilers:

- Improving module-level configuration to serialize configuration along with module CMIs.
- Finer-grained control over caller-side checking.

The initial level of effort in both compilers was reasonably high, as there was some refactoring needed to get all evaluation semantic determination passed through a common function that inspected the

configuration. The second major level of effort was to lazily store the results of that lookup so it was done only when needed, and only once. Once that refactoring was complete, however, adding each new axis of configuration was a very quick effort.

Integration with the implementation of [P3097R3], to support contract assertions on virtual functions, also took some extra considerations resulting in treating the seemingly caller-side static type check as callee-side. This was actually very simple due to how virtual function checking was implemented (with static forwarding functions) but makes it clear that the right way to think of true caller-only checking is as a redundant check a caller might ask for, which is distinct from a caller-facing check on virtual dispatch where that is the primary venue for that assertion to be evaluated.

Integration with [P3400R4] was straightforward as well, capturing the group names and allowed semantics at contract parsing time so that they could be used when the evaluation semantic was needed. In both cases the details needed to be figured out early as doing a constant evaluation to compute groups and allowed semantics when doing codegen for the contract assertions was potentially problematic.

5 Conclusion

The configuration file specification explored here is a starting point that we hope to collaborate on in order to define something that meets all of our needs and allows for robust and portable configuration of contract assertions.

Most importantly, by doing this now we also will have enabled satisfying a vast range of use cases that have previously not been directly addressed by ad-hoc configuration options that initial contracts implementations inevitably begin with.

Acknowledgments

Thanks to Corentin Jabot, Nina Ranns, and Eric Fiselier for earlier reviews of the ideas in this paper and earlier drafts of this paper.

Claude (Anthropic) was used for editorial assistance during the preparation of this paper, as well as significant parts of the prototype implementations.

Bibliography

[P3097R3] Timur Doumler, Joshua Berne, and Gašper Ažman, “Contracts for C++: Virtual functions”, 2026
http://wg21.link/P3097R3

[P3400R4] Joshua Berne, “Controlling Contract-Assertion Properties”, 2026
http://wg21.link/P3400R4