Static Analysis of Contracts with P2900

Document #: P3386R0
Date: 2024-10-15
Project: Programming Language C++
Audience: EWG (Evolution)
Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

Some have suggested that P2900 is unsuitable for static analysis and that we must, therefore, pursue extensive changes to the language to achieve some perceived level of safety when using Contracts. In this paper, we explore some of those suggestions, suggest better alternatives, and present real-world experience showing that static analysis greatly benefitted by the features provided by [P2900R9] contract assertions.

Contents

Revision History

Revision 0

- Original version of the paper for discussion by EWG

1 Introduction

Contract assertions, as developed collectively by SG21 and detailed in [P2900R9], capture algorithms whose purpose is to identify when a program is correct. The C++ Standard, whose aim is to specify how C++ programs are translated and what happens when they are evaluated, must necessarily focus on the behavior contract assertions will have when evaluated and the various interactions they will have at compile time with other parts of the language. This need leads to a predominant focus on the use of contract assertions for runtime checking of correctness since that specification provides the foundation on which all other uses of Contracts will build.

Static analysis is, however, another major use case for contract assertions that does not involve evaluating them at run time. Identifying at run time when a contract assertion is violated is critical to improving program correctness; identifying such violations at compile time reduces the time for realizing improvements even more significantly. More notably, static analysis can identify edge cases that might be missed by tests and rarely seen in production, allowing bugs to be fixed even sooner.

Static analysis of this sort has been employed on C++ code for ages. Modern C++ compilers do significant amounts of static analysis at compile time, to both improve code generation and produce diagnostics. Compilers are already aware of some current macro-based assertion facilities. Other tools perform similar and deeper analysis identifying violations of the preconditions of the core language itself. Finally, existing static-analysis tools have explored identifying violations of macro-based contract-checking facilities in multiple, varying ways.

In this paper, we will elaborate on a few points related to static analysis.

- We will give an overview of what mental models of contract assertions should be applied when analyzing a C++ program that contains them.
- We will describe our experience with applying existing static-analysis tools to codebases that contain macro-based contract-assertion facilities and how those tools will apply even more effectively in the presence of a language-based Contracts facility.
- We will consider whether static analysis would benefit from putting further limitations on what can be placed in a contract assertion or changing the behavior of the language when they occur during the evaluation of a contract-assertion predicate.

By the end of this paper, we should have clearly explained that contract assertions as presented in [P2900R9] provide a powerful tool for enabling static analysis to improve the safety and correctness of all software to which it is applied.

2 What Is Static Analysis?

A correct program is one that achieves its programmer's intent. All programmers, therefore, always strive to write correct programs, almost by definition.

When possible, the language makes writing correct programs easier and incorrect programs more difficult. On the other hand, because so much of a programmer's intent is *not* captured in the code, a programming language cannot guarantee that only correct programs can be written.

The Contracts facility offers a large step toward bridging that gap via the introduction of a mechanism whose primary intent is to identify which behaviors of a program are correct. Many of the complexities of [P2900R9] focus on handling how an incorrect program is mitigated — through invocation of a contract-violation handler or terminating more aggressively.

Static analysis is our general term for any process that attempts to identify whether a given program will be correct without actually executing that program. Any given static-analysis procedure will have a number of different properties.

- The question of correctness can be answered *globally* for a complete program on a specific platform or *locally* for just part of a program. In general, any analysis will make assumptions about behaviors at the boundaries of what is being inspected, such as functions not in the same translation unit meeting their advertised contracts.
  - Even a supposedly global analysis will still be making *some* assumptions that the platform on which the program will run will itself behave correctly.
  - While global analysis might be viable for small programs or for those with unbounded resources ready to devote to verifying program correctness, any useful tool will require the ability to perform local analysis and chain together reasoning when composing a program from individual translation units.
  - The ability to annotate function declarations (not just definitions) with contract checks is essential to performing local analysis without needing to see the full definition of every function under analysis.
- Any analysis will produce information about whether the program being analyzed will be correct for certain hypothetical inputs. This analysis might produce a *false positive* by identifying as incorrect a case that is actually correct or a *false negative* by failing to identify incorrect programs.
  - Any analysis that might produce false positives demands a corresponding escape hatch to suppress those reports once manual analysis has proven the report to be false. On the other hand, such escape hatches are brittle and are not often revisited to ensure that unrelated changes have not turned them into real bugs now hidden by the escape hatch.
  - An analysis with no false negatives cannot be considered *proof* that a particular program has no defects of the form for which the analysis is searching because defects outside the scope of the analysis might always cascade into a violation of that proof.
- Analysis can be done *portably*, aiming to determine if a program is correct with respect to any potential implementation of the C++ abstract machine, or *platform specifically*, considering

the parameters of a specific machine. Even an analysis aiming for portability might make some assumptions, such as the sizes of various built-in types, that restricts the results to commonly available modern hardware and operating systems.

  - The least portable analysis might inspect a single generated program binary, determining correctness for only that singular build configuration.
  - The other extreme would identify correctness for source code that is independent of platform, any preprocessor directives, or even compiler flags.

- When analyzing a program containing contract assertions, an important new variation must be considered: what implementation-defined choice of contract-assertion behavior has been selected. An analysis might be *semantic independent* and aim to prove a program correct independently of the chosen semantics of the contract assertions evaluated, or it might be *build specific*, producing a result for only a single build configuration.
  - As described in [P2900R9], as long as a contract assertion is not *destructive*, the correctness of a program evaluating it is independent of whether it is checked (by definition). Any static analysis might, therefore, assume checks are not destructive and then proceed to be semantic independent. A separate analysis of just the contract-assertion's predicate can then be done to identify potentially destructive predicates.
  - For a platform providing S different semantics with which contract assertions can be compiled and a program in which code is generated for contract assertions in N different locations, a semantic-independent analysis says something useful about S^N different real programs that could be built. This vast number of programs includes, importantly, the program in which all contract assertions have the *ignore* semantic and the full range of programs in which all contract assertions are *enforced*.
  - A build-specific analysis might still consider every single contract assertion's correctness and do so with the knowledge that some will be checked and some will not be checked in the final build. Knowing when an unchecked contract assertion would still be violated if checked is an important concern for determining when a program is correct, in particular when doing a build-specific analysis of the build that ignores all contract assertions.

- The analysis of contract assertions also need not be applicable to *all* contract assertions. A tool might be selective in determining which contract assertions it understands.
  - Tools that wish to track the state of *shadow variables* to do range analysis or similar verifications might limit the contract assertions they consider to only those that perform simple operations on built-in types, e.g., integer arithmetic and comparisons, pointer comparisons, and so on.
  - A tool might extend such analysis to function invocations where it can see the definition of the function.
  - Opaque functions invoked by a contract-assertion predicate can be assumed to be independent of the values being tracked. This form of purity is contextual and can be separately proven once the definitions of a function in a contract assertion are visible.

    - When, in the future, the ability to specify contract assertions that are *never* evaluated at run time is introduced, static analysis can consider pseudo-predicates that are bespoke to that analysis and do not have runtime-implementable equivalents. See [P1728R0] for more discussion on such possibilities or [P2755R1] Section 2.1.14 for how we might build support for this approach on top of [P2900R9].

- Static analysis also exists in a variety of contexts.
    - Third-party tools can be applied at all scales to produce reports that are processed asynchronously to the normal programming workflow and that result in improved software.
    - Analysis tools can be integrated into an IDE, heavily emphasizing those forms of analysis that can be computed in real-time as programmers make edits to their software.
    - Within a compiler, significant amounts of static analysis is performed to optimize a program and produce warnings about commonly suspect behavior.
- Finally, no tool can prove the complete correctness of a program.
    - Any tool will be considering certain properties and attempting to validate that a program locally or globally satisfies those properties and *only* those properties. All assumptions a tool makes will be in relation to those properties.
    - Tools might be verifying memory safety, lack of violation of certain contract assertions or any contract assertions, or satisfaction of specific requirements, such as not making use of the heap.

With all the above variations, static analysis is clearly a broad space with many different constituents, goals, requirements, and capabilities. Contract assertions can help advance the capabilities of almost all such tools.

3 Existing Practice with Assertions

Given the long history of the C `assert` macro, many static-analysis tools already apply logic to assertions with the understanding that they are defensive checks of correctness.

- In JetBrains' CLion's DFA (Data Flow Analysis), assert macros are used and assumed to be true, resulting in warnings based on the information gathered from those assertions. For example, in the code below an assertion's presence will be used to produce a warning on the `if` statement:

  ```
  #include <cassert>

  void f(int p) {
    assert(p > 7);

    if (p == 6)  // CLion displays: "Condition is always false".
      ;
  }
  ```

  Along with the warning comes tools to remove the unnecessary `if` statement entirely.

- Synopsis, in their static-analysis tools, also makes use of the C assert macro and some user-defined macros for which they have implemented bespoke support. Under normal analysis, they assume that assertions are true and use that information to produce better diagnostics. They also provide a checker to analyze the expressions used within assertions to verify that they do not have side effects.

- PVS-Studio also makes use of assert macros when performing static analysis. A key example of where this helps their users is in the removal of false positives by asserting information that the tool itself might otherwise warn as potentially erroneous:

  ```
  #include <cassert>

  //V_ASSERT_CONTRACT // <=

  struct Base
  {
    virtual ~Base() = default;
    int i;
  };
  struct Derived : Base
  {
    float f;
  };

  void foo(Base *p)
  {
    auto q = dynamic_cast<Derived *>(p);
    assert(q != nullptr);
    (void)q->f;  // no warning
  }
  ```

  Due to the presence of the assertion, a false positive is removed, and the debug build, when deployed, will help verify that the assertion itself is sound.

- Within Bloomberg, we have worked with CodeQL to integrate Z3 from Microsoft to do code flow and range analysis to validate contract-assertion correctness. This process begins with a tool that builds a "spec database" of information about the contract assertions (expressed using the `BSLS_ASSERT` family of macros) that exist within the implementations of functions in a library. Once that database is available, analysis of users of that library verifies that assertions of the forms understood by the analysis will not fire.

  Primarily this inspection involves a range analysis applied to integers and pointers and then run through Z3 to identify potential violations.

- The Clang family of compilers understands a special attribute, `analyzer_noreturn`, whose primary purpose is to be placed within assertion macros to indicate that their violation-handling functions should be treated as `[[noreturn]]` when doing static analysis. This annotation allows analyzers to discard branches in which an assertion would fire and thus to reduce the false positives that might result from analyzing paths that are intentionally not supported.

Based on the experiences with the above tools, the overwhelming needs expressed for static analysis fall into a few categories:

- The limitations of a macro-based facility — in particular, that after preprocessing no evidence remains that an assertion is present — make it difficult for static-analysis tools to leverage assertion macros in all builds.
- Not having preconditions and postconditions visible to callers without seeing function bodies is highly limiting to scalability. This lack of visibility led to the building of a bespoke tool to gather spec databases in our work with CodeQL, and other tools often fail to leverage assertions that are not present in inline functions.
- Finally, understanding arbitrary assertion macros is beyond the capability of static analysis. Many provide ways for a macro-based facility to integrate with certain static-analysis tools, such as Clang's `__attribute__((analyzer_noreturn))`, but the need for such collaboration between the static-analysis tools and providers of assertion macros can severely hinder the use of both. More feature-rich assertion macros that provide advanced configuration and flexibility in the choice of mitigation strategies only make implicit integration harder for new tools to provide.

Amazingly, all the concerns above completely go away with [P2900R9]. Contract assertions are not macros, are always present, can be placed on function declarations visible to all translation units, and will be standardized.

4 Unnecessary MVP Modifications for Static Analysis

Changes to the Contracts MVP could be considered to facilitate even more static analysis. Of course, any such change must be measured in terms of how much it limits a user's ability to express checks of the correctness of their programs and which forms of static analysis (of the many listed earlier) will concretely benefit from such a change.

4.1 Defining UB within Contract Assertions

One design goal proposed in [P3285R0] was to prevent or define as much undefined behavior as possible during the evaluation of a contract-assertion predicate. The claim is made in [P3362R0] that this removal is necessary and sufficient to achieve a proof of correctness using static analysis.

Consider, for example, making the choice that integers have wrapping arithmetic. A fairly simple integer-arithmetic based precondition one might want to express is that three integers are positive and sum to a value less than 1000:

```
void f(int x, int y, int z)
  pre( x > 0 && y > 0 && z > 0 )
  pre( x + y + z < 1000 );
```

With normal C++ `int` arithmetic, the above expressions are well defined for only exactly the intended values — `x`, `y`, and `z` in the closed range $[1, 998]$ and where their sum is in the range $[3, 1000]$. Should these values be used to index into an array or as bounds for an iteration, being in the intended range is very important.

When a static analyzer sees the expressions above and knows that integer overflow is undefined behavior, it can come to significantly better conclusions about the expected bounds being demanded of the arguments to `f` and, more importantly, can then produce diagnostics on any values that might be outside those ranges. In particular, given that the first preconditions restrict all the arguments to be positive values, the addition operation combined with a comparison can lead to an immediate reduction of that range to one that is comparatively small and even independent of the other arguments. Without needing to evaluate the addition, for example, any call where `x`, `y`, or `z` might be greater than 998 can be immediately flagged as problematic.

Now consider what values will satisfy the above preconditions if we instead define integer arithmetic as wraparound arithmetic, i.e., if we make integers model $\mathbb{Z}_{2^{32}}$ instead of modeling $\mathbb{Z}$. In such a case, both static analysis and a user must now consider a vastly wider set of values that might satisfy the condition `x + y + z < 1000`. Just as an example, the call `f(1431655932, 1431655932, 1431655932)` would satisfy the above preconditions, though a child will easily tell you it shouldn't and would need to learn a significant amount of advanced math to understand why it might. If we cannot expect our computers to do better than our first graders at basic math, we have failed on a fundamental level.

Any other model in which we alter the actual algebra that is modeled by `int` will have an equally surprising and disastrous impact on the result produced by surprisingly simple contract-assertion predicates.

Of course, one might look at the above call and claim that leaving matters alone would instead make the contract-assertion evaluation itself have undefined behavior. This design, however, has served C++ fairly well since inception and comes with an important and often forgotten benefit: Undefined behavior can itself be made into any behavior, including detecting it as a contract violation. This entire idea is explored more thoroughly in [P3100R1] and below in Section 5.1.

4.2 Removing All Side Effects from Contract Assertions

Now consider another proposal from [P3285R0], which is to render impossible having side effects in a contract-assertion predicate outside its cone of evaluation.[1]

First, a property of this sort — that side effects do not happen outside the cone of evaluation of a function invocation — is not a recursive property. Consider, for example, the following function:

```
bool somethingAboutValuesBetween(int a, int b)
{
  if (a > b) { std::swap(a,b); }

  // Compute a value knowing that a <= b.
}
```

Nothing about the above function is even passed a reference to something outside its cone of evaluation. On the other hand, `swap` unconditionally modifies functions outside *its* cone of evaluation. To truly identify whether a function has this property, one must analyze the full definitions of all

[1]Note that at this point we will not go into the insufficiency of limiting contract-assertion predicates to allow no side effects since that would preclude the ability to allocate, acquire locks, or trace function calls with logging from within any function invoked from a contract-assertion predicate. See [P1670R0] and [P2712R0] for more discussions on this topic.

functions invoked that are not known to have this property, or one must be restricted to avoiding most forms of idiomatic C++.

The situation gets worse for member functions. If we allow non-`const` member functions or `const` member functions that might modify a mutable member, some mechanism must be available to prevent invoking such functions on objects outside the cone of evaluation of the contract-assertion predicate. To provide such a mechanism, significant analysis would need to be done within a contract-assertion predicate to track the flow of values. Consider, for example, the following precondition:

```
struct S {
  mutable int d_x;

  bool increment() const { ++d_x; return true;}
};

S* f(const S& s) {
  return const_cast<S*>(&s);
}

void f(const S& s)
  pre( s.increment() )        // probably outside cone of evaluation
  pre( (&s)->increment() )    // maybe?
  pre( S().increment() )      // inside!
  pre( f(S{})->increment() ) // unknown without definition of f
  pre( [](
    S a;
    return ((S{}.increment()) ? &a : &s)->increment();  // getting harder
  )
}
```

Clearly none of the above examples are good code, but to guarantee a lack of modifications, a mechanism to track values and disallow invalid ones must be fully specified that allows useful code to be written without ever allowing a modification outside the cone of evaluation. Deeper analysis indicates that such proof essentially precludes the ability to use a member function that modifies any members of an object.

Without the ability to use member functions, contract assertions reduce to only the most trivial expressions in terms of built-in types. Such restrictions would prevent contract-assertion predicates from taking advantage of the ability of C++ to encapsulate and leverage abstraction and would be a massive restriction on widespread adoption of the use of Contracts.

4.3 Evaluating Symbolic Predicates at Run Time

Another suggestion from [P3285R0] is to enforce object lifetime safety by requiring that a predicate `object_address` "holds" for a pointer whenever a pointer is used within a contract assertion or a conveyor function (the special kind of functions that are the only kind of functions usable from within contract assertions). The problem here, however, is that contract assertions state something about a single point in time and do nothing to guarantee any stability to the results they are asserting.

Now, such a predicate could have its value inserted by a compiler in many places where its result can be determined statically. Consider, for example, a function that has, as a precondition, a predicate using the symbolic function `object_address` to declare that it is passed a pointer to an object within its lifetime.

```
void f(int* p) pre( object_address(p) );
```

As stated, `object_address`, which can be evaluated at run time as part of the evaluation of a contract-assertion predicate, conservatively determines if the pointer passed to it denotes an object within its lifetime. In particular, if the compiler cannot prove that a pointer will always point to an object within its lifetime when the predicate is evaluated, it must instead evaluate to `false`.

For simple cases in which an object is passed directly from an automatic variable whose address has never escaped, this might be fruitful:

```
template <typename T>
void g()
{
  T x{};
  f(&x);
}
```

Above, we can see that `x` could not have had its lifetime ended prior to the invocation of `f`, so the precondition of `f` will not detect a violation. Now, however, consider the case where we call `f` *two* times:

```
template <typename T>
void g()
{
  T x{};
  f(&x);
  f(&x);
}
```

Is the second call to `f` going to pass its precondition assertions regardless of how `f` is actually defined? If `object_address` is to produce no false negatives, then this obviously cannot be the case if, for example, `f` were defined like this:

```
template <typename T>
void f(T* p) pre( object_address(p) )
{
  static int i = 0;
  if ((++i % 2) == 0)
  {
    p->~T();
  }
  else {
    new (p) T{};
  }
}
```

With this definition of `f`, the above function `g` has perfectly well-defined behavior, but the second call to `f` is made with a pointer to an object outside of its lifetime. Given that `f` *might* be defined in this manner, a conservative definition of `object_address` must fail on the second call to `f`.

This same issue arises when an opaque function is invoked and there are pointers for which `object_address` might *hold* but whose values might have escaped the function. Consider another case where we try to call `f` twice, with two different pointers:

```
template <typename T>
void h(T* p1, T* p2)
  pre( object_address(p1) )
  pre( object_address(p2) )
{
  f(p1);
  f(p2);
}
```

Again, the first call to `f` might get past its precondition given the precondition of `h`. The second call to `f`, however, has no way to know that the first call to `f` did not destroy or delete the object denoted by `p2`, so the second call to `f` will violate its precondition.

Even if we were to consider relaxing the requirements, we can clearly see how easily an assertion can be made that a pointer is valid at one point and quietly destroy it before anything can be done usefully with that assertion if we consider a function that attempts to guarantee that it *returns* a pointer to an object within its lifetime:

```
template <typename T>
T* produce()
  post(r : object_address(r) );
```

When invoking such a function with no parameters, nothing else is evaluated between the call to `produce` and whatever comes after, and thus the validity of the object carries forward as far as the precondition assertions in the functions we described earlier.

On the other hand, consider a function with just one argument, even where that argument has a trivial destructor:

```
int* produce(int i)
  post(r : object_address(r) );
```

The caller of `produce` cannot assume that the definition of `produce` is not one like this:

```
int* produce(int i)
  post(r : object_address(r) )
{
  return &i;   // At least this will warn you, but it's well formed.
}
```

The postcondition in this case is certainly not violated; the function parameter `i` is still within its lifetime when the postcondition of `produce` is evaluated. On the other hand, as soon as control returns to the caller, the object `i` can end its lifetime, and the symbolic function `object_address` no longer holds. Therefore, without seeing a definition of `produce`, a compiler must assume that `object_address` does not hold any time an object of similar type ends its lifetime.

Similarly, if a parameter of `produce` has a nontrivial destructor, that call might invalidate *any* pointer returned from `produce` regardless of its type:

```
struct S {
  ~S();  // in another translation unit
};
int* produce(S s)
  post(r : object_address(r) );
```

No assumption can be made that `~S()` does not free the dynamically allocated integer value that was returned by `produce`.

A compiler could, in theory, analyze much deeper into the definitions of functions and determine, in some additional cases, that `object_address` holds slightly further than the semantics of local analysis could conclude. Having a built-in language feature depend so strongly on quality of implementation, however, would be an awful user experience and would be untenable to use at scale.

An external static-analysis tool can look at assertions like these and layer on arbitrary extra assumptions — in particular, that functions won't invalidate pointers they have no direct interaction with. Yet that is an assumption a tool can be guided into making but not one that we can codify in the Standard itself.

In general, a predicate like `object_address`, which cannot give accurate results without massive instrumentation, is far more appropriate for *unchecked* contract assertions that are never evaluated at run time. Such assertions, explored in [P1728R0] and parts of [P2755R1], can be stated by programmers and then verified as appropriate by tools that understand them with much more flexibility for false positives and negatives, all with a wide range of freedom for quality of implementation.

Having the semantics of a function we provide in the Standard that *must* be used at run time be so fragile and so dependent on quality of implementation would, however, make the entire system completely unusable.

4.4 Restricting Contract Assertions to Marked Functions

Another suggestion from [P3285R0] is to restrict contract-assertion predicates to a set of functions that have been specifically marked to be usable in that context, which will then follow special rules that might reduce the chance of writing destructive contract-assertion predicates. In that particular example, the rule was to require that all functions used be marked with a new attribute, `[[conveyor]]`, that imposed new meaning for undefined behavior within the function and restricted the ability to make modifications to any values outside the cone of evaluation of the function.

This restriction comes with major drawbacks.

- Any member function that does modifications must, by definition, make modifications to the object on which it is applied. This requirement means either that the rules for modification must be lax enough to be meaningless or that all meaningfully non-`const` member functions must not be conveyor functions. In the latter case, not being able to modify objects would prevent almost all use of the abstractions of C++, limiting contract assertions to only the most trivial of predicates involving built-in types.

- Introducing a new annotation that must be present on a function implicitly prevents any currently written functions from being used in contract-assertion predicates. Worse, as soon as changes in behavior occur under the restricted rules, such as the defining of undefined behavior proposed in [P3285R0], users will be required to write two versions of many functions, increasing work and bifurcating the library ecosystem. Such a burden is untenable.
- Static analysis is more than capable of identifying those contract-assertion predicates that meet its requirements and ignoring those that do not, so limiting the Contracts feature to only those predicates that a particular static-analysis tool insists on needing disenfranchises both all more flexible tools and everyone writing runtime predicates that do not fit in a narrow scope.

Overall, pursing the idea of introducing a new kind of function and restricting contract-assertion predicates to only those functions is overly restrictive and provides questionable long-term benefits, and no clear evidence has shown that the idea is worth pursuing.

5 Future Evolutionary Paths

The Contracts MVP provides a strong foundation for features that will allow us to evolve into a language with even greater support for runtime checking, static analysis, safety, and correctness. Nothing discussed in this section is precluded or hindered by the design in [P2900R9], and ongoing effort is being exerted to move toward providing these features in the future.

5.1 Integrating Contract Checking with the Core Language

Static analysis must contend with not only analyzing contract assertions, but also analyzing that users are properly making use of the C++ language itself. As explored in [P3100R1], many core-language constructs that have undefined behavior can instead be clearly phrased as having preconditions that must be satisfied, and such preconditions can be treated in exactly the same way we treat precondition assertions on any function invocation.

More importantly, we can assign distinct semantics — *ignore*, *enforce*, or *observe* — to these preconditions wherever they are encountered and still maintain the same program meaning modulo what is currently undefined behavior. Once tools become capable of applying such a transformation, we also gain the ability to apply the transformation conditionally and, in particular, to apply it when evaluating expressions within a contract-assertion predicate.

With such options, let's recall our earlier example of a function that does some simple integer math in its precondition assertions:

```
void f(int x, int y, int z)
  pre( x > 0 && y > 0 && z > 0 )
  pre( x + y + z < 1000 );
```

With the ability to flag integer overflow as a contract violation, we can turn the call `f(1431655932, 1431655932, 1431655932)` into a contract violation *within* the evaluation of the second precondition assertion of `f`. This functionality requires no special language rules to enable since the compiler is free to do anything when integer overflow such as this occurs.

For functions invoked from a contract-assertion predicate, this solution is slightly less viable, though still possible. Binaries would have to be produced with alternate entry points that do such checking, which would potentially lead to significant code bloat. Many might consider the safety gained for contract assertions worth the cost, and again the benefits of this approach all come with no need for a single additional word in the C++ Standard to enable them.

We can consider again an example shown in [P3285R0]:

```
int f(int x) { return x + 100; }
int g(int a) pre(f(a) > a);
```

Given the example above, invoking the function `f` with a value of `INT_MAX-90` will result in integer overflow during the evaluation of the precondition of `g`. In such a case, when inlining `f` today, a compiler might remove the precondition entirely since the only cases where it fails are those in which the evaluation has undefined behavior. With a better approach to handling undefined behavior in contract-assertion predicates, however, the above function, when inlined, could detect the overflow and produce a contract violation, detecting a bug and allowing the user to fix the newly discovered problem.

5.2 Separately Verifying that Predicates Are Nondestructive

The most promising aspect of contract assertions to static analysis is that they place preconditions and postconditions on declarations, which enables local analysis that is simply not possible (barring alternative methods to provide similar annotations) with C++ today.

Even more powerful, however, is the ability to use the full extent of the C++ language to encode the checking of correctness of a program. Rather than being limited to a bespoke expression language, contract assertions enable users to write checks using the same powerful abstractions available to them when writing the rest of their programs.

By combining the two, static analysis takes advantage of every check that a user can write, even when analyzing only a single translation unit. Many questions, however, might arise as to what possibilities exist in terms of unwanted side effects when invoking an opaque function from within a contract-assertion predicate. If such a predicate does mutate state that is relevant to the correctness of a program, the analysis must then contend with identifying the correctness of the S^N possible different program states that result from S potential semantics chosen for N contract-assertion evaluations.

On the other hand, simply assuming that contract-assertion predicates are *not* destructive reduces this space of S^N states down to exactly 1. While such an assumption might seem bold, in practice we have also seen that writing nondestructive contract assertions is remarkably easy (especially with the benefit of so-called `const`-ification to minimize the risk of unwanted side effects, as explored in [P3261R1]).

The assumption that contract-assertion predicates are not destructive, like all other assumptions on which local analysis is made, can also be proven in a local fashion and then combined with other local proofs to achieve a whole-program correctness proof.

This can't be stressed enough: Concerns about a predicate being destructive can be addressed by analyzing the predicate itself. Once the concerns are addressed for all predicates in a program, correctness of each function can be analyzed on the assumption that predicates are not destructive, and by combining the two analysis results, we can achieve a robust proof of program correctness.

5.3 Identifying Uncheckable Contract Assertions with Labels

[P2755R1] introduces many use cases for a concept called *labels* that allows users to introduce new identifiers that can be placed within a contract-assertion specifier to alter its behavior.

Primarily, labels are needed to allow for in-source control of how semantics are selected when evaluating a particular contract-assertion predicate. Control over semantics comes in two forms: providing compile-time (`consteval`) functions that select a semantic for the evaluation of a particular contract assertion and providing a *set* of allowed semantics for a particular contract assertion. This second feature enables the creation of labels that preclude *all* runtime-checked semantics, and when that happens, we open the door to a large number of possibilities that greatly benefit static analysis.

- Symbolic predicates, such as `object_address` mentioned above, can be used freely within contract assertions whose predicates will never be evaluated at run time. Cases in which a definitive value cannot be determined are freely ignorable by a compiler, while static analyzers will have available to them the full freedom to introduce additional background assumptions to make such predicates even more useful.

- Because such predicates are only hypothetically evaluated but will never directly impact the state of a program, they may freely be expressed in terms of functions that make modifications to program state. This means that we could consider removing `const`-ification from such predicates to more easily facilitate expressing contract checks that can only be accomplished destructively.

- Again, because predicates are not evaluated, meaningful yet destructive contract-assertion predicates can be written without concern for breaking the state of a program. Consider, for example, a function that takes an iterator pair that must have at least three elements in its range:

  ```
  template <typename Iter>
  void f(Iter begin, Iter end)
    pre uncheckable ( std::distance(begin,end) >= 3 );
  ```

  Without the `uncheckable` label above, the precondition will be destructive for any input iterator since the call to `std::distance` will consume the entire range. Even for forward iterators, the above precondition may have unacceptable complexity guarantees for `f` to meet its own contract. Hypothetical analysis that is not done at run time, however, can make use of the precondition to identify the required results of the iteration through the range without actually performing such evaluations.

6 Conclusion

In this paper, we have identified the wide scope of possibilities for static analysis as well as the potential for contract assertions, as provided for by [P2900R9], to facilitate even more useful results from many of those forms of static analysis.

Most importantly, we hope we have made abundantly clear that changes to the Contracts MVP are neither needed nor desirable to have robust and effective static analysis built on top of the Contracts facility. The existing tools that inspect assertion facilities and the possibilities for future features to grow the usefulness of such tools further are each strong indicators that the Contracts facility is on the right track.

Acknowledgments

Thanks to Timur Doumler for help in assembling information regarding the application of existing static-analysis tools to macro-based assertion facilities and to Anastasia Kazakova, Phillip Khandeliants, Charles-Henri Gros, Peter Martin, and Julien Vanegue for their responses about how specific tools approach this topic.

Thanks to John Lakos and Timur Doumler for feedback on this paper.

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

Bibliography

[P1670R0] Joshua Berne and Alisdair Meredith, "Side Effects of Checked Contracts and Predicate Elision", 2019
`http://wg21.link/P1670R0`

[P1728R0] Andrzej Krzemieński, "Preconditions, axiom-level contracts and assumptions – an in depth study", 2019
`http://wg21.link/P1728R0`

[P2712R0] Joshua Berne, "Classification of Contract-Checking Predicates", 2022
`http://wg21.link/P2712R0`

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts Facility", 2024
`http://wg21.link/P2755R1`

[P2900R9] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
`http://wg21.link/P2900R9`

[P3100R1] Timur Doumler, Gašper Ažman, and Joshua Berne, "Undefined and erroneous behaviour are contract violations", 2024
`http://wg21.link/P3100R1`

[P3261R1] Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024
`http://wg21.link/P3261R1`

[P3285R0] Gabriel Dos Reis, "Contracts: Protecting The Protector", 2024
`http://wg21.link/P3285R0`

[P3362R0] Ville Voutilainen, “Static analysis and ’safety’ of Contracts, P2900 vs. P2680/P3285”, 2024
http://wg21.link/P3362R0