Document Number: P1290R2
Date: 2019-02-20
Reply to: J. Daniel Garcia
e-mail: josedaniel.garcia@uc3m.es
Audience: EWG / CWG

Avoiding undefined behavior in contracts

J. Daniel Garcia
Computer Science and Engineering Department
University Carlos III of Madrid

Ville Voutilainen
Qt Company

Abstract

The current wording for contracts opens some opportunities for undefined behavior. Those opportunities derive from the freedom for assuming a contract even when the program is built in a mode where the contract has not been checked. This paper proposes a minimal set of changes to solve the issue by requiring that only contracts that have been checked can be assumed. Additionally, this paper proposes an additional build option to decide whether axioms are assumed or not. Finally this paper also proposes simplifications on the ability to continue after a contract check has failed.

1 Summary

This paper proposes the following changes to the current wording for contracts:

- Clarifies assumption of `default` and `audit` contracts by ensuring that a contract that is not evaluated cannot be used for any kind of assumption. Morever if such contract has been evaluated it can be assumed only when the continuation mode has been disabled.
- Adds a new *axiom mode* that is used to determine if axioms may be assumed or not.
- Removes the ability to enable as a build option the continuation after a contract has failed. In explanation it removes the continuation mode.
- Allows the possibility that a specific contract check can be specified (by stating so in code) to continue in case of failure.

2 Changes from previous version

2.1 Changes in R2

The following major changes are made with respect to P1290R1 [1].

- Added a new section on continuation after contract violation (Section 6).

- Revised text in proposed solutions section (Section 7) with new simplifications on semantics.
- Added a new section comparing with other related proposals (Section 8).
- Revised answers to some question (Section 9).
- Revised proposed wording (Section 10).

2.2 Changes in R1

The following major changes are made with respect to P1290R0 [2].

- Removed references in axioms to *assumption barriers*.
- Revised text in proposed solutions section (Section 7) including the summary of semantics.

3 Introduction

Consider the following code:

```
int f(int x)
  [[expects: x>0]]
{
  return 1/x;
}

int g(int x)
  [[expects audit: x==2 || x==3]]
{
  return f(x);
}
```

Under the current wording [3], contracts that are not checked can be assumed to be true. Consequently, we might have the following behaviors:

- When build mode is **audit**, both checks are intended to be evaluated and consequently assumed. As a consequence `f()`'s precondition can be elided even its checking is on as it can be assumed to always be satisfied.
- When build mode is **default**, only the check in `f()` is intended to be evaluated. However, assuming the precondition in `g()` implies that the precondition inf `f()` is always satisfied and so the check can be elided even though it is on. Consequently, invoking `g()` with a value of `0` would lead to undefined behavior.
- When build mode is **off**, no check evaluated. However, they are both assumed leading again to undefined behavior.

Additionally, when the continuation mode is **on** (continue after violation) the assumption brings in additional undefined behavior as now we are assuming conditions that might be false (because the failed and we returned after running the violation handler).

This original intent of allowing assumptions was to provide an ample margin for optimization. However, the above example is an illustration on how assumptions may lead to undefined behavior.

The interactions of contracts and undefined behavior have been explained in detail in [4]. However, it should be noted that the main goal of contracts is allowing to write more correct software by helping to detect more programming errors. Introducing new undefined behavior was an unintentional effect that needs to be avoided.

Of course, a secondary effect of contracts is giving compilers leeway to perform optimizations. The aim is to satisfy this goal only in the cases that the first goal is not sacrificed.

4 Potential for undefined behavior

In this section we analyze some examples of possible undefined behavior.

4.1 Example 1

In [4] an example provided by Herb Sutter and prototyped by Peter Dimov (see at `https://godbolt.org/g/7TP7Mt`) with simulated contracts is presented.

Essentially this example translated into contracts syntax would be:

```
void f(int x)  [[expects audit: x==2]]
{
  printf("%d\n", x);
}

void g(int x)  [[expects: x>=0 && x<3]]
{
  extern int a[3];
  a[x] = 42;
}

void foo();
void bar();
void baz();

void h(int x)  [[expects: x>=1 && x<=3]]
{
  switch(x) {
    case 1: foo(); break;
    case 2: bar(); break;
    case 3: baz(); break;
  }
}

void test()
{
  int val = std::rand();

  try { f(val); /*...*/ } catch(...) { /*...*/ }
  try { g(val); /*...*/ } catch(...) { /*...*/ }
  try { h(val); /*...*/ } catch(...) { /*...*/ }
}
```

4.1.1 Current status

With the current definition of contracts, compiling the code with the build mode set to **audit** is not problematic. The precondition at `f()` is checked and it can be assumed to be true in next calls. Then, calls to `g()` and `h()` can optimize out the contracts under the assumption that `x` is `2`.

However, if the build mode is set to **default**, the precondition at `f()` would not be checked, but still assumed. Consequently, after the call to `f(val)` the compiler would be allowed to assume that `val` is `2` and the preconditions of `g()` and `h()` would be assumed to be correct and optimized out. This would lead to undefined behavior. For calls to `h()` it might be the case that we got the surprising effect that no function is called. But even worse, if the switch is implemented as a jump table and the compiler assumes the contract and elides the jump table bounds check, then a wild branch would arise. The generated code would attempt to read out-of-bounds at `__jmptbl[val]`, reinterpret whatever bytes it finds there as an address of executable code, and jump there to continue execution. This would result in random code execution and a very serious security issue.

4.1.2 Avoiding unchecked assumptions with disabled continuation

If we change the situation to require that no assumption of unchecked contract can be made when the continuation is disabled, the outcome is quite different.

Compiling the code with the build mode set to **audit** would not be problematic and would lead to the same outcome than with the current wording.

When the build mode is set to **default**, the precondition at `f()` would not be checked and would not be assumed. Consequently, after the call to `f(val)` no assumption can be made on the value of `val`. The preconditions of `g()` and `h()` would not be optimized out and the checks would be performed. No undefined behavior happens.

4.1.3 Avoiding all assumptions with enabled continuation

If we avoid all assumptions when continuation is enabled, we also avoid the possible undefined behavior derived from assuming a failed contract.

4.2 Example 2

This example is a variation of previous example, which is also discussed in [4]. In this variation the precondition at function `f()` is now moved the be an axiom.

```
void f(int x)  [[expects axiom: x==2]]
{
  printf("%d\n", x);
}
```

With the current definition of contracts, axioms are always assumed.

When the build mode is set to **default**, the precondition at `f()` would not be checked (as it is an axiom), but still assumed. In this case, the contract elimination is considered to be intentional as an axiom is considered to be always true.

When the build mode is set to **off**, no precondition is checked, but `val==2` is still assumed.

However, in some cases it might be interesting to be able to remove assumptions introduced by axioms. That would be the case, in a debug version where the developer wants to remove all possible assumptions. On the other hand, there are cases where axioms are desired to be used as assumptions. We consider this aspect orthogonal to the checking level induced by the build mode.

4.3 Example 3

Consider now this simple example:

```
void f(int * p)  [[expects axiom: p!=nullptr]]
{
  if (p) g();
  else h();
}
```

When axioms are assumed `f()` would be optimized to always call `g()`. That is not always desirable. Again the ability to control independently whether axioms are assumed or not gives us what we need.

5 Assumptions and continuation mode

5.1 Basic example

Consider now the following code

```
void f(int * p)  [[expects: p!= nullptr]]
{
  if (p) g();
}
```

5.1.1 Disabled continuation and check default contracts

If we compile with continuation mode set to **off** (the handler never returns) and the checking level is set to **default**, the compiler can use the information from the precondition. The generated code would be essentially the following:

```
void f(int * p) {
  if (p==nullptr) {
    __invoke_handler(); // Never return
  }
  else {
    g();
  }
}
```

The assumption that `p` is not `nullptr` is derived from the structure of the generated code and no special provision is needed to state that the contract is assumed.

5.1.2 Enabled continuation mode and check default contracts

If we compile with continuation mode set to **on** (the handler might return) and checking level set to **default**, the compiler would generate a different code structure that would be essentially:

```
void f(int * p) {
  if (p==nullptr) {
    __invoke_handler(); // May return
  }
  g();
}
```

In this case, the contract (`p!=nullptr`)) is checked, but if it fails is not assumed. Again, no special provisions are needed, the behavior is the consequence of the structure of generated code.

5.2 Another example

Let's try another example:

```
void f(int i) {
  [[assert: i==0]]
  [[assert: i>=0]]
  g();
}
```

5.2.1 Disabled continuation and check default contracts

In this case, the generated code would be equivalent to:

```
void f(int i) {
  if (i!=0) __invoke_handler(); // does not return
  else {
    if (i<0) {              // Always false as i==0 is always true
      __invoke_handler(); // does not return
    }
    g();
  }
}
```

The second check can only be called if the first one was successful. But if the first was successful `i` must be `0` and the second one will be optimized out. Note, that again this is a consequence of the generated code structure, and the resulting code would be similar to:

```
void f(int i) {
  if (i!=0) __invoke_handler(); // does not return
  else {
    g();
  }
}
```

5.2.2 Enabled continuation and check default contracts

In this case, the generated code would be equivalent to:

```
void f(int i)  {
  if (i!=0) __invoke_handler(); // may return
  if (i<0) __invoke_handler(); // may return. Never optimized out
  g();
}
```

Now, both checks are independent and no one can be elided.

5.3 Consequences

As it has been shown through examples, no special provision in the standard is needed to state our goal. Under disabled continuation mode a contract check implies its assumption. Under enabled continuation mode a contract check does not imply any assumption at all.

6 Continuation mode

The ability to enable or disable the continuation mode of a translation unit as a whole is more than it is needed. This has been agreed by several authors of the original contracts proposal. In fact that can be safely removed as long as there is a way to state that a specific check should continue upon violation.

6.1 Reasons to simplify

The original reasons [5] for including in the contract system a mode where execution continues after running the violation handler were:

- Gradual introduction of contracts in old code bases.
- Test of the contracts themselves.

From those, the second one is not a real issue as there are multiple solutions. The real motivation for continuation is the gradual introduction of contracts in old code bases. Besides that, it is also important to address the case where a library that already has some contracts need additional contracts to be added to it. In both cases, being able to add new contracts where a violation only triggers some mechanism to log the violation is important.

For the very same reason, it seems that the current solution (where all contracts in a translation unit change their behavior in case of violation from terminating the program to continue after violation) is not the right solution to the problem. What it is really needed is a mechanism to log violations to new contracts, while preexisting and well checked all contracts remain terminating upon violation.

Consequently, the first thing to do is to remove the continuation build mode from the contract system. The build mode may be `off` (no contact is checked), `default` (only default contracts are checked) or `audit` (both default and audit contracts are checked). If any of those checked contracts is violated, the violation handler is executed and when completed the program is terminated.

To solve the need of contract violation logging a different mechanism could be defined. However, we could also add a new contract construct to keep integration in the contract facility. We propose to add a behavior adjective to any `default` or `audit` contract to indicate that violation implies running the violation handler and resuming execution. For that purpose we propose to use the adjective `continue`. In the next subsection, details are given.

6.2 A simplified model

Once the continuation build mode is removed a translation is controlled by the following options:

- Build mode: `off`, `default`, `audit`.
- Axiom mode: `off`, `on`.

In addition we propose that any precondition, postcondition or assertion may have an optional mode.

```
void f(int * p) {
  g(p);
  [[assert: p!=nullptr ]]; // Default. Terminates on violation.
  [[assert audit continue: *p==42]]; // Continue on violation.
  //...
}
```

Note that `continue` does not specify a level, and in fact it is orthogonal to it. It just changes the behavior of a contract check making it to continue after executing the violation handler instead terminating program execution.

The contract mode `continue` can be applied to any `default` or `audit` contract.

```
[[assert: p!=nullptr ]]; // Default contract. Terminate on violation.

[[assert continue: p!=nullptr]]; // Default contract. Continue on violation.
[[assert default continue: p!=nullptr]]; // Same as above

[[assert audit: p!=nullptr ]]; // Audit contract. Terminate on violation.
[[assert audit continue: p!=nullptr]]; // Audit contract. Continue on violation
```

However, a mode is nonsens for an `axiom` contract as they do not have run-time semantics.

```
[[assert axiom continue: p!=nullptr]]; // Error.
```

We selected the word `continue` as we beleive it is the simplest way of stating programmers intentions (continue execution after this check regardless violation status). We find that adding adjectives (e.g. *maybe*, *always*, *never*) are just confusing and missleading. We could also have added a `terminate` or `exit` mode to state the opposite, but we beleived that this was an unnecessary complication.

7 Proposed solutions

7.1 Avoiding the undefined behavior

This paper proposes to avoid the undefined behavior by clarifying the semantics of every build mode in regards of both evaluation of conditions and assumption of those conditions. For assumption of conditions the clarification needs to address the case where continuation is disabled and when the continuation is enabled.

To define such semantics, the following simple principles are proposed to be followed:

- A contract that, in a given build mode, is not evaluated cannot be used for any kind of assumption. This leads to modes where only contracts that have been checked are used for assumptions and avoiding in this way the identified paths towards undefined behavior.
- Moreover, contracts that have been checked can only be assumed if the continuation mode has been disabled. Otherwise, such assumptions cannot be made. Note, that no special provision is needed as the application of general rules of conditional statements would derive this behavior as illustrated in previous sections.
- Axiom contract are considered as if they had been evaluated when they are enabled. Otherwise, they are ignored. Note that in any case, they need to have a valid syntax,

although expressions in an axiom are allowed to contain invocations to declared but not defined functions. If an axiom contains any invocation that is declared but not defined the axiom is ignored.

Below, the exact semantics of each build level are identified when they are applied to each contract level.

7.1.1 Build mode

The build mode can be any of the following four: **off**, **default**, **audit**. This build level affeccts which checks are evaluated at run-time.

contract-level	Build mode		
	off	**default**	**audit**
`axiom`	not checked	not checked	not checked
`audit`	not checked	not checked	checked
`default`	not checked	checked	checked

Note, that only `audit` and `default` checks are affected by the build mode.

7.2 Allowing assumption of axioms

This paper proposes that an axiom mode is added. The axiom mode can be either **off** or **on**.

- When the axiom mode is **off** an implementation is not allowed to make any assumption.
- When the axiom mode is **on** an implementation is allowed to assume the axiom.

7.3 Continuation mode

This paper proposes to remove the continuation mode.

- An invocation to a returning violation handler results in a call to `std::terminate` after executing the violation handler.

7.4 A new syntax for continuation

This paper proposes to add syntax for a new `continue` adjective that can be applied to any `default` or `audit` contract.

[[**assert continue**: predicate]]; *// Default predicate with continuation*
[[**assert default continue**: predicate]]; *// Same as above*

[[**assert audit continue**: predicate]]; *// Aduit predicate with continuation*

7.5 Summary of semantics

In this section a summary of the build options and their semantics is presented.

The translation is controlled by the following options:

- Build mode: `off`, `default`, and `audit`.
- Axiom mode: `off`, `on`.

1. Build-mode=`off`, Axiom-mode=`off`.
    - No check is performed.
    - No assumption is made.
2. Build-mode=`off`, Axiom-mode=`on`.
    - No check is performed.
    - Checks with `axiom` level are assumed.
3. Build-level=`default`, Axiom-mode=`off`.
    - Checks with `default` level are evaluated and assumed.
    - Checks with `audit` are neither performed nor assumed.
    - Code for `default` contract checks assumes that the violation handler does not return.
    - Code for `default continue` contract checks does not assume that the violation handler does not return.
    - No `axiom` is assumed.
4. Build-level=`default`, Axiom-mode=`on`.
    - Checks with `default` level are evaluated and assumed.
    - Checks with `audit` are neither performed nor assumed.
    - Code for `default` contract checks assumes that the violation handler does not return.
    - Code for `default continue` contract checks does not assume that the violation handler does not return.
    - Checks with `axiom` level are assumed.
5. Build-level=`audit`, Axiom-mode=`off`.
    - Checks with `default` level are evaluated and assumed.
    - Checks with `audit` level are evaluated and assumed.
    - Code for `default` and `audit` contract checks assumes that the violation handler does not return.
    - Code for `default continue` and `audit continue` contract checks does not assume that the violation handler does not return.
    - No `axiom` is assumed.
6. Build-level=`audit`, Axiom-mode=`on`.
    - Checks with `default` level are evaluated and assumed.
    - Checks with `audit` level are evaluated and assumed.
    - Code for `default` and `audit` contract checks assumes that the violation handler does not return.
    - Code for `default continue` and `audit continue` contract checks does not assume that the violation handler does not return.
    - Checks with `axiom` level are assumed.

8 Comparison with other proposals

In this section, we hilight our differences with other related proposals.

8.1 Differences with P1429R0

In P1429R0 [6] four semantics are defined (*ignore*, *assume*, *check-never-continue*, and *check-maybe-continue*). We compare here with those semantics:

- The *ignore* semantics is obtained in our current model by disabling contract checking (build mode `off`).
- The *assume* semantics is obtained for `axiom` contracts by enabling axioms. We have already expressed in this paper the dangers of assuming contracts for `default` and `audit`.
- The *check-never-continue* is what we call termination and it is what happens for any checked `default` or `audit` contract that has no `continue` adjective.
- The *check-maybe-continue* is what we call `continue` and it is what happens for any checked `default` or `audit` contract that has a `continue` adjective.

P1429R0 then proposes the ability to select any of those semantics for any contract level. Of course, all the combinations that we identified in this paper can be expressed by those means. Below we show the equivalence:

- Build-mode=`off`, Axiom-mode=`off`.
  - `axiom`=*ignore*, `default`=*ignore*, `audit`=*ignore*.
- Build-mode=`off`, Axiom-mode=`on`.
  - `axiom`=*assume*, `default`=*ignore*, `audit`=*ignore*.
- Build-mode=`default`, Axiom-mode=`off`.
  - `axiom`=*ignore*, `default`=*check-never-continue*, `audit`=*ignore*.
- Build-mode=`default`, Axiom-mode=`on`.
  - `axiom`=*assume*, `default`=*check-never-continue*, `audit`=*ignore*.
- Build-mode=`audit`, Axiom-mode=`off`.
  - `axiom`=*ignore*, `default`=*check-never-continue*, `audit`=*check-never-continue*.
- Build-mode=`audit`, Axiom-mode=`on`.
  - `axiom`=*assume*, `default`=*check-never-continue*, `audit`=*check-never-continue*.

Note that the semantics *check-maybe-continue* (for simplicity *continue*) is not in any of our combinations, as this semantics, in our model, is only for in-code continuing contracts.

Our most important divergence here is that we consider that individual mapping of semantics to levels is highly dangerous and error-prone. As a mere example, consider the following:

```
void f(int * p)
  [[expects: p!=nullptr]]
  [[expects audit: *p > 0]];
```

If we allow any arbitrary combination we could select, for example, the following semantics: `default`=*ignore*, `audit`=*check-never-continue*. This combination, would lead to undefined behavior when `p==nullptr`.

Our second divergence is that P1429R0 allows either a contract-level or a contract-mode, where the only proposed contract-mode for now is `check_maybe_continue` that is assumed to have a `default` level. Instead of that, we propose de `continue` adjective that can be applied either to `default` or `audit` contracts. We see our model as a more generalized one, and at the same time, we provide a simpler syntax.

Last but not least, P1429R0 proposes explicit syntax for the other three semantics. Adding a syntax for *ignore* makes a contract only syntactically checked. Adding a syntax for *assume* would lead to non-ignorable assumptions. Adding a syntax for *check_never_continue* would introduce non-ignorable terminating checkes. The paper proposes that most uses of those syntaxes would be in combination with macros, which we see as a drawback.

8.2 Differences with P1421R0

In P1421R0 [7] the five semantics from P1333R0, are cited. The main problem with this is that despite the efforts made, it does not seem possible to tell the difference between *check-maybe-continue* and *check-always-continue*. Additonally, the rest of semantics do not need to be defined in the standard unless they are effectively used.

Secondly, P1421R0 proposes an alternate syntax for postconditions for stating the return value:

[[**ensures**(r): r>0]]

The rationale for that is being able to tell the difference between a level and a variable name. We consider that not necessary as there is a fixed number of levels and a developer may easily select any other name for the return value.

Additionally, the proposal for additional arbitrary tags does not seem justified and is a complication that seems a redesign of contracts. This also applies to the ability to add additional implementation defined extensions.

9 Some questions

Why do we add the axiom build mode? Contracts with levels `default` and `audit` are assumed only if they have been checked. That is pure consequence of their evaluation. However, for contracts with level **axiom** some systems may want to take the option to still assume them when other checks are disabled (new proposed axiom mode **on**) while in other systems the policy may be to avoid assuming any axiom when other checks are enabled (new proposed axiom mode **off**).

Would it make sense not to assume axioms when other checks are enabled? An axiom is expected to be used for checks that may be assumed to be true and do not need to be checked. A set of axioms can be inconsistent; if a tool detects inconsistency in a set of axioms that is considered to be an error. Usual practice should be enable assumptions on axioms. However, in some debug modes developers might want to avoid those assumptions. That is obtained by disabling axioms with axiom mode set to **off**.

Why not controlling individual semantics for each contract level? That would lead to some combinations that may be problematic or with surprising behavior. Consider for example, enabling checking for `audit` contracts, but disabling checks for `default` contracts. In other

cases, the combination of choices might even lead again to the undefined behavior that we are trying hard to avoid. Even worse we might end up with the practice of needing to duplicate a predicate in more that one assertion to guarantee it in multiple build modes.

Why not more build modes? The proposed modes seem useful for a variety of use cases. They also seem enough to gain experience with the feature in C++20. After that, if needed, the catalog of build modes might be extended in C++23.

Has this been implemented? The prototype implementation at `https://github.com/arcosuc3m/clang-contracts` implements currently the new proposed semantics for `audit`, `default` and `off` as there is no specific assumption enforcement.

How should axioms be taught? Axioms should be used only for predicates that are never wrong. In fact, that is the mathematical notion of a logical axiom (*a predicate that is universally true*). Axioms are not really preconditions, postconditions, or assertions but a portable way of spelling assumptions. Note that an axiom should never be wrong because the consequence is to inject undefined behavior.

10 Proposed wording

In this section a (probably incomplete) wording is presented. This will be refined before the Kona meeting.

10.1 Part I: Avoiding undefined behavior

In section [**dcl.attr.contract.check**]/**4**, edit as follows:

> 4. ~~During constant expression evaluation (7.7), only predicates of checked contracts are evaluated. In other contexts, it is unspecified whether the predicate for a contract that is not checked under the current build level is evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined.~~ Only predicates of checked contracts under current build level are evaluated.

10.2 Part II: A new axiom mode

After [**dcl.attr.contract.check**]/**4**, add a new paragraph:

> 5. A translation may be performed with an *axiom mode* which can be either *off* or *on*. Unless the *axiom level* is *off* it is unspecified whether the predicate for an *axiom* contract is evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined.

10.3 Part III: Removing continue build mode

Remove completely [**dcl.attr.contract.check**]/**7**.

Add a new clause in [**dcl.attr.contract.check**].

> 7. After completing the execution of the violation handler execution is terminated by invoking the function `std::terminate` (13.5.1).

10.4 Part IV: Adding a contract mode

TBD.

Acknowledgments

The work from J. Daniel Garcia is partially funded by the Spanish Ministry of Economy and Competitiveness through project grant TIN2016-79637-P (BigHPC – Towards Unification of HPC and Big Data Paradigms) and the European Commission through grant No. 801091 (ASPIDE – Exascale programmIng models for extreme data processing).

Acknowledgements to R2

We would like to express our thanks to those who participated on email discussions as well as in WG21 mailing lists.

Acknowledgements to R1

We would like to express our thanks to those who participated on email discussions since previous versions including John Lakos, Gabriel Dos Reis, Bjarne Stroustrup, Alisdair Meredith, Joshua Berne, and Hyman Rosen. They provided good feedback and interesting questions.

Acknowledgements to R0

I would like to express our thanks to John Lakos, Gabriel Dos Reis, Bjarne Stroustrup, Ville Voutilainen for all the feedback provided and interesting discussions with alternate points of view. Herb Sutter helped to improve explanations on the effects of undefined behavior Richard Smith and Jens Maurer provided initial clarifications on the effect of existing wording as well as additiona feedback.

References

[1] J. Daniel Garcia and Ville Voutilainen. Avoiding undefined behavior in contracts. Working paper p1290r1, ISO/IEC JTC1/SC22/WG21, January 2019.

[2] J. Daniel Garcia. Avoiding undefined behavior in contracts. Working paper p1290r0, ISO/IEC JTC1/SC22/WG21, November 2018.

[3] Gabriel dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. Support for contract based programming in C++. Working paper p0542r5, ISO/IEC JTC1/SC22/WG21, June 2018.

[4] Ville Voutilainen. UB in contract violations. Working paper p1321r0, ISO/IEC JTC1/SC22/WG21, October 2018.

[5] Gabriel dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. A Contract Design. Working paper p0380r0, ISO/IEC JTC1/SC22/WG21, May 2016.

[6] Joshua Berne and John Lakos. Contracts that work. Working paper p1429r0, ISO/IEC JTC1/SC22/WG21, January 2019.

[7] Andrzej Krzemienski. Assigning semantics to different Contract Checking Statements. Working paper p1421r0, ISO/IEC JTC1/SC22/WG21, January 2019.