Doc.No.: P3352R0
Project: Programming Language - C++ (WG21)
Authors: Andrew Tomazos, Martin Uecker
Date: 2024-07-08
Audience: EWG, SG22
Target: C++26

Taming the Demons (C++ version) – Undefined Behavior and Partial Program Correctness

We propose changing the specification of undefined operations in C++, such that they do not affect the behavior of defined operations.

Currently, if an undefined operation appears anywhere in the execution graph of a program, all operations in the execution graph (even ones that happen before an undefined operation) have no requirements placed on their behavior. We call this the **abstract rule**. We propose tightening that rule to only place no requirements on the behavior of the undefined operations, leaving requirements on the other operations in place. We call this the **concrete rule**.

At first blush, this proposal seems like a breaking change, as it is commonly thought that there are desirable optimizations that depend on the abstract rule. It turns out, to everyone's surprise, this is not the case. This has been determined both empirically and analytically. Many people have gone looking for desirable optimizations that would be broken by this, and haven't found any. After analyzing what is going on, we now understand why we haven't found any, and have a good explanation for their non-existence.

There are three categories to the misperception:

1. Optimizations that people think are broken by the concrete rule, but are not.
2. Optimizations that people think are allowed by the abstract rule, but are not.
3. Optimizations that people think are desirable, but are not.

Implementations are effectively using the concrete rule already, and are expected to continue to do so in the future. Changing the standard from the abstract rule to the concrete rule is mainly a standardization of the existing practice, and it is an existing practice that implementations do not want to change and cannot practically change even if they wanted to.

Under the concrete rule, the program can produce any behavior during execution of an undefined operation, including optionally finishing by stopping the program (in any manner). So

the concrete rule effectively places no requirements on the behavior of operations that do not happen before all undefined operations (hereafter **subsequent operations**). This is so because whatever the collective behavior of such subsequent operations, that behavior is indistinguishable from the behavior of the undefined operation. This formalism is just saying that upon execution of an undefined operation, the program is put into an unpredictable, non-portable state for the remainder of the program execution until termination. This is so under the abstract rule, and it remains so under the concrete rule:

```
int main()
{
  int a = /*...*/;
  int b = /*...*/;

  log(a / b);   // #1
  log("hello"); // #2
}
```

If b is zero then operation #1 is an undefined operation. The required behavior of operation #2 is to print "hello". If the program printed "goodbye" instead of "hello", then there is no way to distinguish between (a) the behavior of operation #1 was to print "goodbye" and terminate the program; and (b) operation #2 misbehaved and printed "goodbye" instead of "hello". That is, (a) is an allowed possible execution of the abstract machine, and its observable behavior is to print "goodbye" and terminate. A real implementation of (b) produces the same observable behavior as the (a) possible execution of the abstract machine. So the real implementation is conformant, because its observable behavior correctly matches that of an allowed possible execution of the abstract machine. (It is explicitly irrelevant that the underlying "structural" causes are different - all that matters is the observable behavior.)

Let us define **local-unobservable operations** as operations that "happen between" prior observable operations and following undefined operations. More precisely, local-unobservable operations are those that (a) happen before all undefined operations and subsequent operations; (b) do not have observable behavior; and (c) do not happen before an operation with observable behavior that is not an undefined nor subsequent operation (ie excluding those mentioned in (a)). Under the combination of the as-if rule and concrete rule - effectively no requirements are placed on their behavior either. Local-unobservable operations have the same status as subsequent operations. Even if they are transformed by the implementation to produce observable behavior, it is indistinguishable from the observable behavior of the undefined operations and subsequent operations. Thus, they too retain the same status under both the abstract rule and concrete rule:

```
extern int x;

int f(int a, int b)
{
```

```
 x = b ? 42 : 43;
 return a / b;
}
```

Implementations are free to, and do, constant-fold the ternary expression to 42, as-if b is nonzero, because when b is zero, the undefined operation has no requirements on its behavior, so the constant folding cannot be distinguished from the behavior of the undefined operation, in the same fashion as the previous example.

Worth mentioning here is that there is a set of operations with similar properties and status to local-unobservable operations that happen “even earlier”. Of the subset of operations yet unclassified in this paper, they are the ones in that subset that (a) do not have observable behavior and; (b) their effects do not affect the observable behavior of any other operations in the subset. These can be transformed under the concrete-rule/as-if-rule provided that (a) and (b) continue to hold after the transformation. There are no imaginable cases of optimizations that transform these to violate (a) or (b), so we treat them as part of the local-unobservable set:

```
volatile int y;

int f(int a, int b)
{
  int x = b ? 42 : 43; // #1
  y = 3; // #2
  return a / b;
}
```

When b is zero: in the above #1 has a similar status to the local-observable operation from the previous example, but it cannot be transformed to produce observable behavior or alter the behavior of #2 that happens after it and before the undefined operation. Similar constant folding to the previous example, conforms to this. (If #1 where transformed to assign 2 to y, it would not be conforming to the concrete rule, as this operation happens before #2 - but we cannot even imagine an optimization that would do this.)

We are left with finding the status of **observable operations** that happen before all undefined operations. Under the concrete rule they must produce their required behavior, as all defined operations must. But what about under the abstract rule? There are two categories of observable operations: **volatile operations** and **non-volatile operations**. Volatile operations are volatile reads/writes. Non-volatile operations are all other observable operations.

Non-volatile operations are treated by implementations similarly to external function calls, in that they are not inlined (transformed away by the optimizer), even with link-time or whole-program optimization. This is expected, as to produce observable behavior, a program must interact with the external world at runtime in a way that can be observed. Otherwise, there would be no way to falsify that an observable behavior of a real implementation occurred (actual semantics), and

so no way of comparing it to the observable behavior of the abstract machine (abstract semantics). On Unix-like operating systems, this interaction typically occurs through system calls, though all implementations must have comparable mechanisms.

All optimizers in practice treat non-volatile operations with the same abstraction as an external function call, meaning that execution is not guaranteed to continue after them; they mark the boundary of a basic block in call flow analysis. Consequently, operations that happen after a non-volatile operation are not known to be part of the execution graph if and until a non-volatile operation that happens before completes and control returns. Thus, even under the abstract rule, non-volatile operations must adhere to their standard behavior requirements. That is, the implementation cannot assume that undefined operations are part of the execution graph until the non-volatile operation completes successfully.

It is theoretically possible for implementations to use special knowledge of a particular non-volatile operation (that it guarantees to return) to prove that subsequent undefined operations are on the execution graph, and leverage the abstract rule to undefine their behavior. We emphasize again, no implementations do this. No one has or would ask for this. They do not do this because it would not be a desirable nor worthwhile optimization to make. It would also be dangerous. Users want and expect that observable behavior that happens before an undefined operation does what it is required to do - as is the case today:

```
extern int x;
extern void g();

int f(int a, int b)
{
  if (b == 0) g();
  return a / b;
}
```

Under both the abstract rule and concrete rule, the implementation cannot (and do not) dead-code-eliminate the if statement under an assumption that b is nonzero:

- Under the abstract rule, the implementation doesn't know whether the divide-by-zero undefined operation is on the execution graph of the program if and until g returns, and hence whether or not all operations of the program have no requirements placed on them. That is, if b is zero and g does not return, then the program does not have undefined behavior. So prior to g returning you can't assume b is nonzero.
- Under the concrete rule, observable behavior in g must be explicitly preserved (but this is already the case anyway, by the same argument above for the abstract rule).

For the case of **volatile operations** there is a practical difference between the abstract rule and concrete rule. Volatile operations have observable behavior and are usually

implementation-defined to guarantee to return. Volatile operations are the only known case where this proposal has real-world consequences.

First of all, notice that the whole raison d'etre of volatile operations is to suppress local optimization around them. A volatile operation must interact with the outside world in the same fashion as a non-volatile operation (communicating by interacting with memory). For non-undefined-behavior related optimizations, they are suppressed by volatile. That is the whole point of them having observable behavior. To carve out an exception for undefined-behavior related optimizations would be at least inconsistent. Further, it is neither worthwhile nor desirable to do so:

```
volatile int x;

int foo(int a, int b, bool store_to_x)
{
  if (!store_to_x) {
    return a / b;
  }
  x = b; // #1
  return a / b; // #2
}
```

#1 is a volatile operation. If b is zero then under the abstract rule, implementations are free to (and some do) precalculate a/b at the top of the function like so:

```
volatile int x;

int foo(int a, int b, bool store_to_x)
{
  int c = a/b; // #3
  if (!store_to_x) {
    return c;
  }
  x = b; // #1
  return c;
}
```

because under the abstract rule they can assume #3 is not an undefined operation and so call flow will continue on to #1. Under the concrete rule this is not allowed. Note that this would not be the case in the following:

```
extern void g();

int foo(int a, int b, bool store_to_x)
{
```

```
  if (!store_to_x) {
    return a / b;
  }
  g(); // #4
  return a / b;
}
```

Here #4 is a (potentially) non-volatile operation so the main argument applies, and it has the same status under both rules.

This family of optimizations around volatile operations is not worthwhile, not desirable and is inconsistent with other observable behavior (non-volatile operations). The operating system and device driver community (and others that actually use volatile operations) have explicitly requested that these optimizations be banned. Volatile operations should be treated as other observable behavior is.

The above cases are an exhaustive treatment of all possible operations in a C++ program execution graph. Summarizing in “chronological” order:

1. Start of program through observable operations (non-volatile and volatile).
2. Local-unobservable operations (and “even earlier”)
3. Undefined operations
4. Subsequent operations through end of program.

Changing from the abstract rule to the concrete rule has little impact on implementations, present or future - and the concrete rule does not change the status of most operations. So why is this change even worth it?

- The concrete rule is simpler and safer than the abstract rule. It is easier to communicate and reason about. There have been compiler bugs that were caused by the confusing abstract rule. (For example: https://developercommunity.visualstudio.com/t/Invalid-optimization-in-CC/10337428?q=muecker and https://bugzilla.redhat.com/show_bug.cgi?id=520916)
- The concrete rule matches the rules of the C23 standard. C and C++ are front-end languages that are translated into an intermediate form by the implementations. The optimizer operates on the intermediate form, and is agnostic to which front-end language was used. It therefore makes sense to have the same set of rules governing the optimizers behavior, especially given that it will conform to the tighter concrete rule of C23 anyway.
- The concrete rule makes it far easier to show partial correctness of programs with undefined operations. The complex (and clearly counter-intuitive) reasoning given in this paper regarding the abstract rule is inherently hard to follow and work with. The concrete rule is much easier as a basis for proof of correctness/incorrectness of programs and implementations.

- There are desirable real-world consequences for users of volatile operations for this proposal, and these changes have been explicitly requested by them.
- There may have been something we have missed, some desirable optimization that is rendered non-conforming by the change from the abstract rule to the concrete rule. We would much prefer that such optimizations are discovered and investigated so WG21 and implementations can work together to decide if they should be considered standard defects or compiler bugs. The abstract rule makes such discoveries less likely, it defaults to considering these potential compiler bugs as conformant.

We propose the following changes to the C++ standard:

(Wordsmithing note 0: The proposed wording has two main components. (a) We (non-evolutionary) refactor [intro] to more precisely use an "operational semantics" framework. The abstract machine is described in terms of a taxonomy of operations, and operations produce effects and behaviors when they are executed. This forms a foundation on which the second component can be laid. (b) We (evolutionary) change C++ from the abstract rule to the concrete rule by redefining the semantics of undefined operations.)

> [intro.defs]
>
> 3.20 erroneous behavior
> the behavior of an erroneous operation; well-defined behavior that the implementation is recommended to diagnose

(Wordsmithing note 1: We want to link the definition of erroneous behavior to the newly defined and italicized term "erroneous operation", hence the addition.)

> 3.26 implementation-defined behavior
> behavior~~, for a well-formed program construct and correct data~~, that depends on the implementation and that each implementation documents

(Wordsmithing note 2: Implementation-defined behavior can include aspects of the machine that are not constructs or operations. For example, "what constitutes an interactive device" is not a construct. The term "correct data" also doesn't fit in that same fashion, it is at best unclear what it is trying to achieve. Erroneous behavior and undefined behavior are part of a well-formed program too, and yet neither have this phrase. We assert that, at best, the phrase doesn't carry its own weight, so recommend removal.)

> 3.63 undefined behavior
> the behavior of an undefined operation; behavior ~~for which this document imposes no requirements~~ that depends on the implementation

(Wordsmithing note 3: We want to link the definition of undefined behavior with the term “undefined operation”. “Undefined operation” was previously defined implicitly through use, we have now italicized a definition.)

(Wordsmithing note 4: The phrase “for which this document imposes no requirements” is ambiguous. Is it saying that if the document omits a description of the effect of some operation, then that operation is classified as undefined behavior? Or is it saying that if there is an explicit notation that some operation is “undefined behavior”, then the document is giving the implementation permission to do whatever it wants? If the latter, then does that include “time travel optimizations”? We have proposed a much more precise and unambiguous description in [intro.abstract], that intentionally does not include the phrase “imposes no requirements”, and so propose striking that phrase from the above definition.)

(Wordsmithing note 5: As per the proposed [intro.abstract] wording, under the concrete rule, undefined operations become similar to unspecified behavior and become part of the nondeterministic behavior of the abstract machine that admits its set of possible executions. It is thus correct to say and appropriate to say “it depends on the implementation” in the same fashion as the other kinds of behavior that do.)

> 3.64 unspecified behavior
> behavior~~, for a well-formed program construct and correct data,~~ that depends on the implementation; usually one of a set of well-defined behaviors

(Wordsmithing note 6: We recommend the strike for the same reason as wordsmithing note 2. Unspecified behavior consists of both aspects and operations in the same fashion as implementation-defined behavior, and the same argument for its removal applies.)

(Wordsmithing note 7: In order to differentiate the three kinds of behavior that depend on the implementation (implementation-defined, unspecified and undefined), we have the existing phrase on implementation-defined that “each implementation documents”, we add “one of a set of well-defined behaviors” to unspecified, and then through omission of both of these extra phrases on “undefined” we communicate that neither of those restrictions apply to it. The “range” of implementation-defined behavior is communicated in the same fashion, by omission. As is the “documentation requirement” of unspecified behavior, by omission. This makes those two omissions on undefined behavior consistent with that messaging.)

> [intro.compliance.general]
>
> ~~1. The set of diagnosable rules consists of all syntactic and semantic rules in this document except for those rules containing an explicit notation that “no diagnostic is required” or which are described as resulting in “undefined behavior”~~.

(Wordsmithing note 8: The reference to undefined behavior here is malformed. Undefined operations are part of a correct execution of a program, and they do not violate a rule. After

further analysis we concluded that the whole definition of “diagnosable rule” isn’t carrying its own weight given recent changes to the standard, and recommend striking and replacing uses with `rule with an explicit notation that “no diagnostic is required”`)

> 2. Although this document states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:
>
> 2.1. If a program contains no violations of the rules in [lex] through [thread] as well as those specified in [depr], a conforming implementation shall accept and correctly execute[3] that program, except when the implementation's limitations (see below) are exceeded. [Note: A “correct execution” includes the execution of any erroneous or undefined operations. See [intro.abstract] and [intro.execution]. -end note]

(Wordsmithing note 9: This is a rework of a former footnote into an informative note, as it is something that can be potentially confusing and surprising, so we wanted to give it more prominence. We reworked the wording to use the terms erroneous operation and undefined operation and updated the references to the italicized definitions.)

> 2.2. If a program contains a violation of a rule ~~for which no diagnostic is required~~ with an explicit notation that “no diagnostic is required”, this document places no requirements on implementations with respect to that program.

(Wordsmithing note 10: The first change was treated in wordsmithing note 8. The pluralization of requirements is a little editorial fix that we tacked on, it seems like more correct English than the singular. It should be noted that a program that is “ill-formed; no diagnostic required” due to, for example, linker errors like ODR violations - is completely unrelated to this proposal, unrelated to operations and unrelated to undefined behavior. This proposal has zero impact on IFNDR.)

> 2.3. Otherwise, if a program contains
>
> 2.3.1. a violation of any other ~~diagnosable~~ rule,

(Wordsmithing note 11: The strike is from wordsmithing note 8. The addition of “any other” is not strictly necessary here because this is guarded by an If/Otherwise, but we felt that it helps clarity when read in isolation, that this is referring to a subset of rules.)

> …
>
> ~~FOOTNOTE 3) “Correct execution” can include undefined behavior and erroneous behavior, depending on the data being processed; see [intro.defs] and [intro.execution].~~

(Wordsmithing note 12: See wordsmithing note 9)

[intro.abstract]

1. The semantic descriptions in this document define a parameterized nondeterministic abstract machine. This document places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.

2. Certain aspects and operations of the abstract machine are described in this document as implementation-defined behavior (for example, sizeof(int)). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects. Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).

3. Certain other aspects and operations of the abstract machine are described in this document as unspecified behavior (for example, order of evaluation of arguments in a function call ([expr.call])). Where possible, this document defines ~~a~~the set of allowable behaviors of each unspecified behavior. ~~These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.~~

(Wordsmithing note 13: As part of applying the concrete rule, we have combined undefined operations into nondeterministic behavior of the abstract machine. Thus, the struck sentence has been moved and integrated below into a new paragraph. After the move, the new final sentence "Where possible, this document defines a set of allowable behaviors." seemed to require more clarity (too vague and nonspecific) hence the added editorial wording to make the sentence more clear and to standalone in its intent.)

4. Certain operations of the abstract machine produce *observable behavior*. These are behaviors that:

- control interactive devices (for example, displaying output or awaiting and accepting input);
- write data to files;
- access volatile glvalues; or
- have other observable effects.

What constitutes an “interactive device”, and the list of “other observable effects”, is implementation-defined.

(Wordsmitting note 14: We have moved the definition of observable behavior above its usage in the as-if rule. We have rewritten the definition of observable behavior to describe it as a unit of behavior of an operation, rather than the totality of all the observable behavior of the execution of a program. This sets up the modification of the as-if rule to refer to the observable behaviors (plural) of the program and the order of those behaviors. This granulation and ordering of observable behaviors is necessary to express the concrete rule. Conveniently, it also allows the previous special case ordering that was tacked on to the old definition of observable behavior to be refactored into the as-if rule much more neatly. It also makes the “minimum requirements on an implementation” now a direct consequence of the as-if rule combined with the implementation-defined list of “other observable effects”.)

~~4. Certain other operations are described in this document as undefined behavior (for example, the effect of attempting to modify a const object).~~

~~[Note 1: This document imposes no requirements on the behavior of programs that contain undefined behavior. — end note]~~

5. *Undefined operations* are operations of the abstract machine:

- described with an explicit notation that their “behavior is undefined”; or
- for which a description of their effect is omitted in this document.

(For example, an attempted modification of a const object.)

Upon control passing to an undefined operation, the abstract machine enters an unpredictable state for the remainder of program execution. While in an unpredictable state, the abstract machine is allowed to produce any (possibly empty, possibly infinite) sequence of observable behaviors. Execution in an unpredictable state is allowed to either continue indefinitely or to terminate the program. [Note: It follows that undefined operations cannot affect observable behaviors that happen before them. Assumptions implied by potential undefined operations cannot “reverse time travel” and alter an observable behavior that happens before. -end note]

(Wordsmithing note 15: The first sentence of this paragraph is purely editorial, we introduce “undefined operation” as a formally defined term and establish how operations are specified as being undefined operations in the standard (the specification technique). The second sentence is the heart of the concrete rule. It explicitly normalizes the range of allowable behaviors of undefined operations, and, importantly, that these behaviors can only be produced upon execution of the operation. The final informative note just clarifies in simple terms the consequence of the concrete rule on the relationship between observable behavior and

undefined behavior. The first sentence is similar in meaning to the clarification that WG14:N3128 applied to C23. The second sentence says the same thing but in terms that optimizer writers use.)

> 6. Unspecified behavior and undefined operations (the resulting “unpredictable state”) are the nondeterministic behaviors of the abstract machine. An instance of the abstract machine can have more than one *possible execution* for a given program and a given input. There is one possible execution for each combination of the allowable behaviors of its nondeterministic behaviors, and that possible execution produces that combination of behaviors. [Note: This document places no requirement on implementations to predict or document which possible execution of the corresponding instance of the abstract machine corresponds to a given program execution on their implementation -end note]

(Wordsmithing note 16: After applying the concrete rule, it turns out that undefined operations become part of the nondeterministic behaviors of the abstract machine (alongside unspecified behavior). This paragraph defines the “nondeterministic behaviors” and the “possible executions”, and specifies how the set of possible executions is derived from the nondeterministic behaviors. This set of possible executions is then used by the as-if rule (Paragraph 8 below). The informative note clarifies that implementations are not required to predict or document nondeterministic behavior - that it may be unpredictable.)

> 7. *Erroneous operations* are operations of the abstract machine described with an explicit notation that their “behavior is erroneous” (for example, reading certain uninitialized variables). This document describes their allowable behavior. In addition to this allowable behavior, each erroneous operation is also permitted to:
>
> - issue a diagnostic upon execution; and/or
>
> - terminate the program at an unspecified time after the operation completes.
>
> Recommended practice: An implementation should issue a diagnostic when an erroneous operation is executed.

(Wordsmithing note 17: We introduce “erroneous operation” as a formally defined term for operations that have erroneous behavior. We felt that this was more appropriate to do as part of the taxonomy of other kinds of operations within [intro.abstract]. There are several places where we refer to erroneous and undefined operations together. Having them consistent makes this easier.)

> [Note: An implementation can issue a diagnostic if it can determine that an erroneous or undefined operation is reachable under an implementation-specific set of assumptions about the program behavior, which can result in false positives. — end note]

8. A conforming implementation executing a well-formed program shall produce the same observable behaviors, and in the same order, as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input.

(Wordsmithing note 18: This change to the as-if rule makes it explicitly clear that the order of the observable behaviors is significant in the comparison of the abstract semantics and actual semantics.)

~~However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).~~

(Wordsmithing note 19: The above is the abstract rule of undefined behavior. As per the evolutionary intent of the proposal, we strike it and have replaced it with the concrete rule added earlier in the wording. See wordsmithing note 15.)

~~If the execution contains an operation specified as having erroneous behavior, the implementation is permitted to issue a diagnostic and is permitted to terminate the execution at an unspecified time after that operation.~~

~~Recommended practice: An implementation should issue a diagnostic when such an operation is executed.~~

~~[Note 2: An implementation can issue a diagnostic if it can determine that erroneous behavior is reachable under an implementation-specific set of assumptions about the program behavior, which can result in false positives. — end note]~~

~~8. The least requirements on a conforming implementation are:~~

~~8.1. Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.~~

~~8.2. At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.~~

~~8.3. The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.~~

~~These collectively are referred to as the observable behavior of the program.~~

~~[Note 3: More stringent correspondences between abstract and actual semantics can be defined by each implementation. — end note]~~

(Wordsmithing note 20: The above requirements have been moved, refactored and reworked into the earlier [intro.abstract] changes. See previous wordsmithing notes.)

This proposal, P3352, is almost entirely derived from [WG14:N3128] which proposed essentially the same thing that P3352 does with the same rationale, but targets C instead of C++. [WG14:N3128] is the result of several years of work of the Undefined Behavior study group of the C standard committee and was unanimously forwarded by it to the main WG14 body, and it has been adopted by the main body into C23. P3352 basically proposes that C++/WG21 "follows suit" with C/WG14 on this issue.

There is a highly related proposal [WG21:P1494], which proposes retaining the abstract rule but adding a new primitive called "observable checkpoints" to provide a selective barrier against it. [WG21:P1494] suggests in part that "we might [instead] choose to adopt the requisite changes to the definition of undefined behavior". These "requisite changes to the definition of undefined behavior" are what P3352 is proposing. In our opinion, P3352 is the safer and simpler choice. The authors of [WG14:N3128] are in agreement with that position. In [WG14:N3128], they state "In our opinion, it seems much simpler and safer [than P1494] to ensure that all observable behavior automatically has this property", referring to the effect of the concrete rule. That is, effectively the C23 concrete rule makes it so that all points in the program are "observable checkpoints" (as they already are in practice), so the term and concept of "observable checkpoint" is not needed under it.

References

[WG14:N3128]
N3128
Date: 2023-05-05
Title: Taming the Demons -- Undefined Behavior and Partial Program Correctness
Author: Martin Uecker

[WG21:P1494]
P1494R3: Partial program correctness
Audience: EWG; CWG; LEWG; SG22
S. Davis Herring <herring@lanl.gov>
Los Alamos National Laboratory
May 20, 2024