Defining Contracts

Document #:	P2358
Date:	2021-04-10
Project:	Programming Language C++
Audience:	SG21 Contracts
Reply-to:	Gašper Ažman <gasper.azman@gmail.com> John McFarlane <john@mcfarlane.name> Bronek Kozicki <brok@incorrekt.com>

1 Purpose of this paper

"When **I** use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean – neither more nor less."

"The question is," said Alice, "whether you **can** make words mean so many different things."

This paper defines terms so that the SG21 group doesn't talk past each other. If you have a problem with a definition in this paper, by all means become a co-author to clarify it. This paper does not take design stances, merely assigns words to concepts any participant in the contracts discussion has tried to make distinct.

This paper extends [[P2038R0]], based on some observations on the telecon.

2 Bugs, damned bugs, and Undefined Behavior

We distinguish quite a few categories of bugs in this group. This paper proposes the following taxonomy.

People on SG21 have historically argued that the distinction is/is not important. This paper takes the stance that if someone considers it important, it's important, and the design should take a stance on how to handle each category. The final decision may be "we handle them uniformly".

2.1 AMVs: Abstract Machine Violations

This contains "hard UB" and other bugs that sanitizers can already trap on today. These are always bugs, have always been bugs, and are already bugs. Currently cause time-traveling optimizations and other things that can make it seem as though source-code doesn't correspond to machine code.

These are bugs because they violate the Language specification.

2.1.1 Atomic AMVs: Atoms of language UB

The obvious stuff: `++INT_MAX`, `*nullptr`, `__builtin_unreachable()`. This is often called "hard UB".

2.1.2 Transitive AMVs: Contract-surfaced Abstract Machine Violations

AMVs, but visible further (across compilation units, peephole limits, …) because a contract specification on the interface exposes the preconditions of the specification (Denotational-semantics-style).

These matter because these are things the compiler *would have optimized on* but didn't get the chance to due to technical issues.

These bugs violate both the abstract machine and the interface specification.

2.1.2.1 Examples

- `[[pre: dst != nullptr]]`
- `[[pre: is_aligned(*p, cacheline_size)]]` on the interface of a function that wraps a shared-library interface.
- `[[pre: survives_callback_time(*callback_pointer)]]` on `void register_async_callback(callback_ptr)`. This kind are **AMVs down-the-line** - won't occur in the function, but will occur later.

2.2 BizBugs: Violations of a Business Specification

These are bugs that *wouldn't* result in a violation of the abstract machine (to orthogonalize the term), but signal business logic bugs. Sending an order with an invalid order ID, or `stable_sort()` not being actually stable are examples of such bugs.

> **Note:** This paper discourages the use of the term **library UB**, because that term contains both *AMVs down-the-line* and *Hyrum's Law-style* violations. Using the term *Library UB* has sucked many an hour of this SG and the authors wish to put a stop to that.

2.2.1 Checkable BizBugs: Contract-surfaced Violations of a Business Specification

These are *BizBugs*, but machine-checkable through a supplied contract specification. `[[post: order_id != "[invalid]"]]` and `[[pre: all_elements_are_unique(first, last)]]` are possible specifications that could surface such bugs.

2.3 HyrumBugs: Use of Unspecified parts of an interface

These are especially insidious, because *the code obviously works, look at the integration tests*, but could stop working if the implementation of the used library changes within the bounds of its published specification. Adding contracts to an interface can expose HyrumBugs and expose the existence of code that works correctly "by accident".

2.3.1 Example

This works on libstdc++ (and is handy for asserting that the guard actually locks the correct mutex):

```
bool locks(std::lock_guard<std::mutex> const& g, std::mutex const& m) {
    return std::bit_cast<std::intptr_t>(g) == std::bit_cast<std::intptr_t>(&m);
}
```

2.4 Contract Bugs: Bugs in contract specifications

These are bugs where the actual specification is wrong. Acknowledging the existence of these is important because we have to have a way of testing for these.

3 Other Phenomena

3.1 Contract Smoothing

Some preconditions are only *AMVs* depending on control flow:

```
int select(bool c, int* x, int* y)
[[pre: x != nullptr]]
[[pre: y != nullptr]]
{
    return c ? *x : *y;
}
```

The contract specification *smoothes* the space of detectable bugs from a conditional:

```
[[pre: c ? (x != nullptr) : (y != nullptr)]]
```

to the saner requirement that both always be non-null. Another example is the contract on `memcpy(dest, src, count)`, where both `src` and `dest` must be valid pointers even if `count == 0`.

Contract smoothing is a source of *HyrumBugs*.

4 Effects of Detected Contract Violations

In this group we've sometimes had trouble agreeing what various terms for "what happens when we detect a contract violation at runtime".

Handlers have been explored in detail in [[P2339R0]]; this paper tries to give unambiguous names for the possible behaviors.

This paper proposes the following taxonomy:

4.1 Undefined Behavior

The compiler is allowed to assume a contract violation won't happen.

```
// <contracts>
#ifdef I_SOLEMNLY_SWEAR_I_TRUST_UBSAN
[[gnu:always_inline]] inline void violation_handler(diagnostic-information) {
    std::assume(false);
    // or
    __builtin_unreachable();
}
#endif
```

and assume link-time optimization.

This is what [[P2064R0]] argues is a bad idea. The authors think allowing this is not a bad idea, but some of them wouldn't use it on production builds.

4.2 Unspecified Behavior

The compiler is not allowed to assume anything about what happens when a contract violation happens.

```
// <contracts>
extern void violation_handler(diagnostic-information);
```

This formulation has interactions with `noexcept` which any design permitting throwing needs to address. Selecting it makes the design a bit more complicated.

4.3 Unspecified but can't throw

```
// <contracts>
extern void violation_handler(diagnostic-information) noexcept;
```

This formulation is the simplest answer to *how do violation handlers interact with noexcept functions* - they never throw. Still allows continuing after a violation.

4.4 Unspecified but never returns

```
extern [[noreturn]] void violation_handler(diagnostic-information) noexcept;
```

This formulation does not admit continuing after a contract violation, if contract checking is enabled. Specifically, it forbids log-only handlers.

4.5 Specified alternatives

These include all proposals where the standard only ships a few predefined violation handlers. All such proposals need to specify the handlers to ship and are thus more complicated than leaving behavior unspecified, but are perhaps simpler for the ecosystem to reason about.

5 References

[P2038R0] Andrzej Krzemieński, Ryan McDougall. 2020-01-11. Proposed nomenclature for contract-related proposals.
https://wg21.link/p2038r0

[P2064R0] Herb Sutter. 2020-01-13. Assumptions.
https://wg21.link/p2064r0

[P2339R0] Andrzej Krzemieński. 2021-03-15. Contract violation handlers.
https://wg21.link/p2339r0