Analysis of Contracts Papers

Document Number:	P4330R0
Date:	2026-07-28
Intent:	Inform
Audience:	WG21

Reply-to:	Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

This paper examines select papers that propose to extend P2900 contracts. Each paper is evaluated through two lenses: whether it harms the profiles safety framework by subordinating it to contracts, and whether it violates the founding design principles of the language. The examination finds that the papers, individually and as a collection, foreclose the design space for profiles before profiles exist, route safety guarantees through a single contract-violation handler that the programmer cannot bypass from source code, and make continuation past undefined behavior the default. This paper asks for nothing.

Executive Summary

The contracts facility takes complete ownership of the safety architecture before profiles are designed, so profiles arrive as a dependent layer with nothing of their own.

- D4314R0 routes a profile's runtime configuration through contracts, leaving profiles able only to subtract from a design they do not control.
- D4315R0 fixes the runtime behavior a profile may use as a rule of the language, removing the possibility of a library-level alternative.
- P3100R8 makes profiles a dependent layer above contracts, denying them an independent guarantee or violation response.
- P3290R6 moves authority over the safety response from source into the build, so a profile's guarantee does not survive compilation.
- P3400R4 assigns the response to core-language undefined behavior entirely to contracts, leaving profiles no independent role.
- P3595R0 anchors safety configuration in contracts and lets the build override any source-level intent.
- P3850R1 routes the violation response through contracts, displacing profiles as an independent facility.
- P4262R0 defines class invariants within contracts, occupying design space that profiles would otherwise own.
- P4275R0 binds safety configuration to contracts with no architectural slot for a separate framework.
- P4298R0 routes the safety-violation response through contracts, foreclosing an independent path for profiles.

If the committee advances these papers, profiles are permanently subordinate to contracts and cannot operate as an independent safety facility.

If these papers move forward, C++ loses the ability to determine what a program does by reading its source.

- D4314R0 moves a construct's behavior into external activation state, violating locality.
- D4315R0 imposes cost on every translation unit whether or not it opts in, violating zero-overhead.
- P3097R3 ties assertion evaluation to runtime dynamic type, violating static reasoning.
- P3100R8 recasts core-language behavior as externally configured assertions, violating source authority.
- P3290R6 routes legacy assertions through a single program-wide handler, violating component autonomy.
- P3400R4 fixes assertion semantics through external configuration and cross-unit coupling, violating local reasoning.
- P3850R1 sanctions continuation past a detected violation, violating fail-safe defaults.
- P4262R0 hides invariant evaluation behind external configuration, violating explicitness.

- P4283R0 extends the facility without field evidence, violating evidence-based design.
- P4298R0 relocates exception behavior into external configuration, violating source determinism.

The committee faces standardizing a language whose behavior lives in build systems rather than in programs.

Revision History

R0

- Original version.

Introduction

The committee voted to include contracts (P2900) in the working draft. That vote settled one question: the language will have a contract-checking facility. It did not settle the architectural question of who owns the response to core-language undefined behavior, whether profiles are an independent safety framework or a configuration layer over contracts, or how runtime safety checking should be configured. Those questions remain open and contestable.

This paper goes through select papers published in the mailing and analyzes their effects on safety in the language and on the profiles safety framework.

D4314R0: Profile runtime configuration owned by the contracts substrate

D4314R0 routes the configuration of a profile's runtime response through the contracts evaluation-mode system rather than an independent profiles mechanism, so the shared substrate owns how that response is configured and profiles are left to subtract from a design they do not control (Section 3, "What a profile can do", item 3). It fixes that response in the language rules themselves, where the response is ignored by default, a profile's activation may only restrict the allowable evaluation modes, and the program is ill-formed when no valid option remains, which places the configuration in the standard and keeps competing approaches from shipping independently as libraries (Section 2, "In short"; Section 3, item 3). By presenting the equivalence of a profile's runtime response with contracts as settled background rather than an open question, the paper also raises the consensus that later giving profiles an independent mechanism would require (Section 3, item 3). This dependence of profile configuration on the contracts substrate is how the contracts program harms profiles.

In D4314R0, what a construct does is not determined by the source at its point of use but by activation state and guidance the user supplies elsewhere. For a construct with erroneous or undefined behavior, a profile redefines that behavior as a change to the base language whose effect is not contingent on activation, so a program that relied on the prior fallback changes behavior with nothing at the site to signal the replacement (What a profile can do, sec. 3; Glossary, sec. 4). Whether a construct is enforced, observed, ignored, or rejected is fixed by whether the profile is active in the surrounding region and by a mode selected from other user guidance, not by anything at the construct, so local reading, code review, and static analysis lose the information that governs behavior (In short, sec. 2; What a profile can do, sec. 3). The added analysis is always

present but set to allow any evaluation mode with the default being to ignore every evaluation, and only activation restricts it toward terminating behavior, so a program performs no verification until the programmer opts in (What a profile can do, sec. 3). The programmer's intent for a construct, whether it enforces or merely observes, cannot be stated at the construct and is instead delegated to separate user guidance, dividing that intent from the construct it governs (What a profile can do, sec. 3). Thus contracts harms the language.

D4315R0: profiles lose library configurability of runtime evaluation semantics

D4315R0 fixes the evaluation semantics that a profile's runtime instrumentation may use as a rule of the language, keyed to whether the profile is active, since an active profile restricts the allowed evaluation semantics and removes otherwise-allowed evaluation modes (D4315R0 sections 7 and 11). Placing that mechanism in the standard rather than in a library means a competing evaluation approach cannot be delivered as a library, and any later adjustment requires a new revision of the standard. A profile leaves its runtime instrumentation in place whether it is active or suppressed and only narrows which evaluation semantics that instrumentation may use (D4315R0 section 11); because activation is selected at build time, identical source produces different runtime behavior across build configurations, and a reader cannot tell from the source what a violation will do. This is a further case of the contracts program harming profiles, converting behavior that could remain library-configurable into a fixed rule of the language.

D4315R0 Section 7 does not preserve the zero-overhead principle, because it treats the redefinition of previously undefined behavior as a case that cannot change on profile activation and sets the bar for that change at an "acceptable" performance impact rather than none. That redefinition therefore takes effect in every translation unit regardless of whether any profile is enabled, so a program that never opts into the profile still pays whatever runtime cost it introduces (D4315R0 Section 7). Code that does not use the feature pays for it anyway, and imposing this cost on every translation unit is one way the contracts program harms the language.

P3097R3: virtual-call assertions depend on runtime type, not the source

P3097R3 makes the preconditions and postconditions evaluated at a virtual call depend on the object's runtime dynamic type, so a reader cannot determine from the code at the point of use which assertions will run, a result the proposed wording's own example demonstrates (Sections 2 and 4.3). It discards the assertion inheritance that overriding functions previously received and proposes no in-language replacement, so an override that needs its base's assertions must restate them by hand, which the paper concedes is impractical and, when a predicate names a private base member, impossible (Sections 4.5 and 5.1). It advances this design without deployment or field evidence, resting on illustrative constructions and committee votes rather than reports from production use (Section 1). Each item removes protection or local reasoning from C++, so the paper extends the contracts program at the language's expense.

P3099R3: user-defined diagnostic messages for contract assertions

This paper does not harm profiles.

This paper resolves its open design questions by moving hazards and withheld capability onto the user of the assertion facility rather than absorbing them into the facility itself.

- The contract-violation-handler interface returns a null pointer when no diagnostic message is supplied, so a handler that logs the returned string without first testing for null dereferences it and crashes the program (Section 2.5).
- Delivery of a written message to the violation handler and to diagnostic output is recommended practice and not a normative requirement, so whether a message reaches the handler is fixed by build configuration and is not determinable from the source at the point of use (Sections 2.4, 6.11.3, and 17.10.3).
- Runtime-generated diagnostic strings are declined on the grounds of security implications the paper states are not fully understood, leaving no supported means to embed the value of a runtime variable in a message (Section 2.2).
- A dedicated second-argument message parameter is added to the assertion facilities in place of the general labels mechanism the paper itself identifies as the most extensible, so the language gains a single-purpose construct where a reusable facility would serve (Section 2.1).

Taken together these decisions enlarge the language with a single-purpose construct, withhold a needed capability, and make an unchecked crash the easy default, so the contracts program leaves the language worse than it found it.

P3100R8: Profiles left dependent on the contract-assertion substrate

P3100R8 places profiles above the contract-assertion machinery as a dependent layer, so that a profile inherits its guarantee, its response to violations, and its configuration from that machinery instead of defining any of them itself.

- A concrete profile is presented as a named configuration preset over the runtime contract-evaluation, erroneous-behaviour, and subsetting tools, with an audit-only variant that renders a program ill-formed when the chosen semantics do not match a profile's stated guarantees, so a profile holds no runtime guarantee of its own and cannot change unless that substrate changes first (Section 4.4, Section 7.2).
- A failed implicit contract evaluation for a null dereference, an overflow, or other core-language undefined behaviour is routed into the same program-wide contract-violation handler used for explicit assertions, and the paper defends that arrangement while describing a per-feature handler as poor design, leaving a profile no independent path for responding to a violation (Section 5.1, Section 5.6).
- The single, program-wide handler is treated as a central aspect of the P2900R14 design, no separate handler is provided for the compiler-inserted implicit evaluations, and users are directed to branch inside the global handler, which forecloses any arrangement in which a profile supplies its own response mechanism (Section 5.6).
- The user-facing safety configuration is anchored in contract-evaluation semantics and in Labels, with a profile expanding into P3400R4 Labels directives, so the configuration design is settled by machinery that is specified and ships first and any later profiles framework must conform to it (Section 4.4, Section 7.2).
- By adding observe-noexcept and enforce-noexcept semantics that call std::terminate rather than propagate an exception, the paper retires the exception-safety objection to continuing after a failed implicit contract evaluation, yet

under the observe semantic with a normally returning handler execution still proceeds past a language-undefined state into unconstrained behaviour (Section 5.5).

Each of these choices settles the underlying substrate before profiles are designed and denies profiles a guarantee, a response, or a configuration of their own, which is how the contracts program harms profiles.

P3100R8 recasts every operation that can exhibit core-language undefined behavior as carrying an implicit contract assertion whose acting semantic is fixed not in the source but by build-time, implementation-defined configuration (Sections 5.1, 5.2, and 5.4).

- The behavior of an operation on a violated precondition, whether it continues, stops, yields an erroneous value, or reverts to prior undefined behavior, is selected by configuration outside the translation unit, so a reader or static analyzer of the source cannot determine what the operation does (Sections 5.2 and 5.6).
- The safe default is discarded because defaulting to the safe behavior is deemed "too user-hostile," so the conforming default proceeds as though every precondition holds and diagnosis of a violation becomes an explicit opt-in (Sections 5.2 and 5.4).
- Every implementation must supply the full runtime violation-handling facility, including handler dispatch, the violation object, and the evaluation semantics, whether or not a given program uses it (Section 5).
- Adopting the chosen resolution of the noexcept interaction requires the compiler to treat every core-language expression that can trigger a violation as potentially-throwing, so programs that never throw from a violation still pay the added code-size and lost-optimization cost (Section 5.5).
- A program-wide handler governs every implicit violation, so separately developed libraries cannot choose their own violation handling and must coordinate through that shared handler and whole-build configuration (Sections 5.2 and 5.6).

This relocation of core-language behavior and safety out of the source and into build configuration and mandatory runtime machinery weakens the language for every program subject to it.

P3290R6: Safety response authority moves into the build

P3290R6 relocates authority over the response to a detected violation from the source into the program-assembly and build steps, so a guarantee expressed in source cannot bind the shipped program.

- The central contract-violation handler is selected by whoever assembles and links the final program, which subordinates a library's or source author's chosen response to a single program-level handler that overrides it at assembly time (Section 2.1).
- Control over the response is framed as a negotiation between only the source author and the build or compiler, each able to override the other, so the model contains no place for a guarantee that survives the build (Section 2.3).
- The paper treats the central handler as the settled destination that every existing assertion facility must route through, anchoring safety response in the contracts substrate as an already-decided premise that later mechanisms would have to displace rather than extend (Section 1, Section 2.1).

- The same assertion expression is given different runtime meaning under different build configurations, so identical source cannot be reasoned about by reading it and instead depends on flags the build system selects (Section 2.2, Section 2.3).
- A noexcept, log-and-continue overload removes the exception-safety objection to executing past a detected violation while leaving the underlying hazard of continuing unaddressed (Proposal 1.3, Section 2.1).

By establishing that safety response is owned by the contracts substrate and configurable only at build time, the paper forecloses the binding, source-level guarantees that profiles depend on, advancing the omnibus thesis that the contracts program harms profiles.

P3290R6 makes the reporting behavior of pre-existing assertions depend on program-wide configuration rather than on anything visible where the assertion is written. The paper routes legacy and manually detected violations into a single program-wide contract-violation handler, so a separately developed component does not select its own violation behavior and instead inherits whatever handler is installed elsewhere in the program (Section 2.1, Proposal 3). Whether the assert macro invokes that handler or prints to stderr is chosen by defining **STDC_WANT_ASSERT_USES_CONTRACTS** at the point of inclusion and, per Section 2.3, by implementation-defined command-line flags, which places the safety-relevant behavior outside the source (Section 2.2). An assert(...) expression therefore looks identical whether or not it integrates with contracts, and a reader consults non-local configuration to learn whether it prints and aborts or invokes the handler (Proposal 3, Section 2.3). The paper also declines a general mechanism in favor of a fixed set of named functions, and it concedes that each future semantic forces new library entry points (Section 2.1). The contracts program weakens local reasoning.

P3400R4: Contracts leaves no independent standing for profiles

P3400R4 assigns the language's response to core-language undefined behavior entirely to the Contracts facility, leaving a separate profiles framework no independent role.

- Evaluation-semantic computation and local-violation-handler dispatch are defined as core-language behavior and depend on a new Itanium ABI entry point, so a competing response mechanism cannot ship as a library and can be altered only by a future standard revision (Sections 3.3.2, 3.3.4, 6, and 7).
- The Introduction frames the facility as a two-party collaboration between the source author and the build engineer, with no place for an independent authority whose guarantees neither party can override, so accommodating such an authority would require redesigning the architecture rather than extending it (Section 1).
- The routing of core-language undefined behavior through Standard Library labels and the build-configuration system is presented as the design with no separate framework weighed as an alternative, so the Contracts substrate owns the safety-configuration design and any later alternative is anchored to a mechanism that has already shipped (Section 4.3).
- The premise that Contracts is the substrate for expressing and mitigating core-language undefined behavior is carried forward as settled background inherited from prior papers and is never reopened, so reopening who owns that response comes to require progressively stronger consensus (Sections 4.3 and 4.5).
- Because the effective behavior of a given core-language operation is selected by the build system through an implementation-defined configured semantic, a reader cannot determine a program's runtime behavior from its source alone (Section 3.3.2, the Glossary, and Section 4.3).

Taken together, these provisions place the response to core-language undefined behavior within the Contracts program across the language, its ABI, and the build system, and thereby harm profiles.

P3400R4 makes the effective behavior of a contract assertion, including whether a violated predicate stops the program, a product of implementation-defined build configuration and cross-translation-unit coupling rather than of the source in which the assertion appears.

- An unlabeled assertion takes an implementation-chosen semantic that is permitted to be ignore or observe, so continuation past a violated predicate is the default, and even a terminating label is permitted to be relaxed to ignore on platforms that cannot enforce it, leaving the safe path opt-in and not guaranteed (Sections 3.4.5, 3.5.1, 6.11.2).
- A label such as review rewrites a terminating configured semantic into observe, so a violated predicate is logged and execution continues past the point the source declares must hold, and the only marker at the call site is an opaque token while the reader is instructed to understand the predicate without the control object that decides continuation (Sections 2.2, 3.3.2, 3.5.1).
- The effective semantic and violation behavior of a labeled assertion depend on implementation-defined build configuration, on the labels seen in other translation units through ODR-equivalence and ABI coupling, and on the program-wide handler, so reading the assertion at the call site does not reveal whether it terminates, observes, or is ignored and static analysis has to look outside the source (Sections 2.3, 3.5.1, 3.5.3).
- Every translation unit that sees a function must carry ODR-equivalent assertion-control expressions and an always-enforced label becomes part of the function's ABI, so a separately developed caller that attaches a different control choice to the same interface is ill-formed or ABI-incompatible (Sections 3.5.1, 3.5.3).
- A library cannot set its own contract behavior in isolation, because the effective semantic is fixed by build configuration that maps group names across the whole build and violation handling still ends at a single program-wide replaceable handler shared by every component (Sections 2.3, 3.3.5).

In place of a facility for writing assertions, P3400R4 supplies a build-and-run system whose behavior no reader can determine from the program and whose default is continuation past a violated predicate, and in that form the contracts program harms the language.

P3595R0: Safety configuration anchored in contracts and overridable downstream

P3595R0 builds a system for selecting contract evaluation semantics entirely within the contracts substrate, and it treats source-level safety intent as something the build-time consumer may override.

- The semantic-selection mechanism is bound to the label mechanism of P3400R4 and treats that substrate as the venue with no alternative considered, so the configuration design is owned by the mechanism that ships first and no independent framework retains a path to define its own configuration (Sections 1, 2.1, 2.2).
- The design is framed as a two-party arrangement between the source-code author and the build-time consumer, with no architectural slot for a party that imposes a non-overridable guarantee (Section 1).

- Ultimate control over runtime response is vested in the consumer who compiles and links, and the paper states that disabling behavior externally to the source code is the only available option, so any guarantee expressed in source is advisory rather than binding (Sections 1, 2.5).
- The same assertion is permitted to mean different things at runtime depending on build configuration, and implementation-defined semantic divergence is expressly sanctioned, so identical source can exhibit different behavior across configurations and toolchains (Sections 1, 2.2, 2.5).

By locating safety configuration inside the contracts substrate and leaving every guarantee overridable downstream, the paper occupies the space an independent profiles framework would need, which confirms the omnibus thesis that the contracts program harms profiles.

The paper locates the selection of a contract-assertion evaluation semantic entirely outside the source, in external JSON configuration files loaded through `-fcontract-configuration-file=` and in command-line flags matched against location, namespace, module, group, and kind, so the intended semantic of a `pre`, `post`, or `contract_assert` cannot be stated in the code itself (Sections 2.1-2.5).

- A reader at an assertion cannot determine from the source whether that assertion is ignored, observed, or enforced, because the semantic is resolved from distant files and flags under a first-match rule, so code review and static analysis cannot establish behavior from the point of use (Sections 2.3-2.5).
- Whether execution continues past a violated assertion, or whether a caller-side check runs at all, is changed silently by external configuration, since the `observe` semantic is a configurable output and the recommended default maps all caller-side checks to `ignore` (Sections 2.2, 2.5).
- Separately developed libraries and translation units cannot independently choose contract behavior, because the same configuration is required across every translation unit and every compiler building the program (Sections 1, 2.5).
- Contract behavior is pushed out of the language into a build-time configuration format and resolution algorithm that implementations supply and users operate, so the specification governs how programs are configured and built rather than how they are written (Sections 2-2.5).

By moving the meaning of an assertion out of the code and into the build, the design removes the programmer's ability to state and read the intended semantic in the source, and in that removal the contracts program works against the language it is meant to serve.

P3850R1: Routing safety response and configuration through the contracts substrate

P3850R1 establishes the contracts substrate as the owner of both the response to core-language undefined behavior and security-critical safety configuration, leaving no architectural room for a separate safety facility.

- The observed evaluation semantic bundles invoking the violation handler with continued execution past an undefined state, so the contested continuation cannot be separated from the uncontested logging (Section 2.4).
- Implicit contract assertions guarding core-language undefined behavior receive the same evaluation semantics as ordinary assertions, so the observe semantic reaches even states such as a null dereference that the language leaves undefined (Section 2.4).

- These implicit assertions are integrated with the global contract-violation handler, making that handler the single point of control for the response to core-language undefined behavior that a separate safety facility cannot bypass (Section 2.4).
- The configuration model admits only the source author and the build, with no architectural slot for a third party imposing non-overridable source-level guarantees (Section 2.2).
- Security-critical safety configuration is routed through the contracts Labels system, and because the roadmap carrying that mechanism is already EWG-approved, an independent framework arrives too late to offer an alternative (Section 2.2).

By placing both the response to undefined behavior and safety configuration inside the contracts substrate, the plan displaces the independent profiles facility, confirming the omnibus thesis that the contracts program harms profiles.

P3850R1 recommends prioritising a group of C++29 contract-assertion extensions whose runtime effect is determined by build configuration and program-wide state rather than by the source, several of which permit execution to continue past a detected violation.

- Implicit contract assertions are inserted by the compiler with no syntax at the point of use and can be observed rather than enforced, so a detected core-language undefined-behaviour condition is reported and execution then proceeds past it, with nothing at the affected line to show that a contract assertion exists (Section 2.4).
- Labels that restrict a contract assertion to ignore and observe only make a configuration in which a violated condition does not halt execution a first-class supported option, and continuation past that condition requires no explicit effort at the point of use (Section 2.2).
- The syntax `pre <my_library | audit> (expression)` leaves the assertion's runtime effect to label definitions and build-time selection, so reading a labelled assertion does not reveal whether it is ignored, observed, or enforced (Sections 2.2 and 2.4).
- Violation handling routes user code into a single standard global contract-violation handler governed by program-wide label configuration, so separately developed libraries do not independently select their own contract behaviour without cross-component coordination (Sections 2.2 and 2.3).
- The roadmap mandates runtime and library infrastructure - a global handler together with machinery to define and combine label semantics - that implementations must provide, extending the facility beyond a language feature toward a runtime system (Sections 2.2 and 2.3).

By moving the meaning of a contract assertion out of the code and sanctioning continued execution past a detected violation, this roadmap advances the contracts program's harm to the language.

P4186R0: multi-year profiles plan committed on sentiment, not field evidence

The Motivation section founds a multi-year plan on committee poll sentiment and on external pressure to treat the language as unsafe rather than on production deployment, field data, or user reports. The same section presents profiles as the best approach the committee knows while acknowledging there is insufficient agreed-on documentation to state what profiles can and cannot do. It records that four prior profiles papers have all failed, yet it commits SG23, EWG, and CWG time through 2029 to the mechanism (Motivation). Committing years of committee time to a mechanism the paper cannot define is how this program harms the language.

P4262R0: Class invariants routed through contracts, foreclosing profiles ownership

P4262R0 defines class invariants entirely as an extension of the C++26 Contracts facility, so a core correctness feature is defined within contracts rather than left open to a profiles-based treatment. The design adds new `std::contracts::assertion_kind` values (`invariant_pre`, `invariant_post`, and `invariant_manual`) and routes every invariant violation through the same contract-violation handler and `std::contracts::contract_violation` object as any other contract assertion, occupying the class-invariant design space before any profiles-based treatment exists (Abstract; Sections 3.4.1 and 3.4.2). It further places all control over the evaluation of invariants inside the contracts machinery, through an assertion-control label drawn from P3400R4 and the ignore, observe, enforce, and quick-enforce semantics, leaving no independent path for a profiles-based framework to own that configuration (Sections 3.4.1, 3.4.2, and 3.6). This extends the contracts program into design space a profiles-based safety framework would otherwise own, closing off that path before it can be proposed.

P4262R0 builds the validation of a type's invariants on the C++26 Contracts facility and makes that validation implicit and non-local, so that the code at the point where an object is used does not reveal whether the object's invariants hold or whether they are evaluated.

- A function that has already broken the object's invariant reads it through its const accessors or re-enters it through internal calls with no validation performed, and nothing at the call site indicates that the object is inconsistent, so an object in a broken state is used with no local signal (3.5.3, 3.5.7).
- The finest-grained variant evaluates a type's invariants on entry to and exit from every function that receives the object, including standard algorithms such as std::sort that legitimately move and swap elements through intermediate states in which the invariant does not hold, so the mechanism does not compose with generic code that has no knowledge of the type (3.5.13, 3.4.4).
- Whether an object's invariants are evaluated as it crosses a boundary, and how, is governed by implementation-defined build configuration, assertion-control objects, labels, and a proposed facet that computes its behavior from the boundary context, none of which appears at a plain call such as g.transfer(...), so a reader cannot determine the behavior from the code at the point of use and both review and static analysis lose the information they need (3.4.2, 3.6).

- The only remedy the paper offers for passing an object through generic code while its invariant does not hold is a mechanically generated parallel projection type that depends on C++ reflection as it develops and on a broad and invasive change to the type system, so the path that would make the feature work with standard algorithms relies on future language and tooling rather than on what a programmer has today (3.5.14).

Grounded in the contracts facility, this mechanism removes from the point of use the ability to tell whether an object's invariants hold, breaks correct generic code, and defers its own repair to language features that do not yet exist, so it makes C++ harder to read, review, and rely on.

P4275R0: Contracts assertion-control leaves no room for a profiles framework

P4275R0 routes safety-relevant configuration through the contracts assertion-control label system, so the facility that ships first owns the safety-configuration design and a later profiles framework cannot evolve enforcement or hardening without first changing contracts (Requiring Enforcement and Library Hardening). The configuration model it specifies admits only two parties, the source author who annotates assertions with labels and groups and the build engineer who selects the semantics, and it provides no architectural slot for a third party that imposes non-overridable, codebase-wide guarantees, so profiles could be introduced only by redesigning the model rather than extending it (Controlling Cost of Checking, Deploying New Assertions Safely, and Groups Configuration Example). The same source expression is given different runtime behavior depending on build configuration, so an assertion's effect cannot be determined from the source alone and the language separates into build-configuration dialects that no fixed profile could hold constant (Library Hardening, Compute Semantic Example, and Groups Configuration Example). By binding the safety-configuration architecture to contracts and leaving no place for a separate framework, the contracts program harms profiles.

The proposed design separates the behavior of a contract assertion from the source that expresses it and places that behavior in label definitions, build configuration, and external files.

- A label rewrites the configured semantic and the review label converts an enforcing semantic into observation, so a failed precondition is detected and execution then enters the function body while the source shows only the label name (Compute Semantic, Deploying New Assertions Safely).
- The starting semantic comes from the build system and the build configuration governs the result, so the same precondition enforces, observes, or does nothing according to the build, and neither a reviewer nor a static analysis tool determines its behavior from the source (Compute Semantic, Controlling Cost of Checking).
- The decision of which semantic each group receives lives in an external configuration file and in build flags while the source only tags an assertion with a group name, so the program's safety behavior is expressed outside the language and apart from the code it governs (Groups - Configuration Example).
- Whether a group, opt, or audit assertion is evaluated is set through build configuration by whoever assembles the final program, and group names share a program-wide space, so a separately developed library does not govern its own assertions and instead depends on the whole program's build and on agreement over shared group names (Grouping Contract Assertions, Controlling Cost of Checking).
- Configuration files that map group names to semantics, a global and chained handler facility, and a staged resolution from configured to computed to effective semantic together specify how a program is configured and built, so every

implementation carries this configuration and resolution machinery whether or not a given program uses it (Groups - Configuration Example, Violation Handling, Compute Semantic).

Relocating the meaning and control of an assertion out of its source and into build and configuration machinery removes local reasoning from the language, which is how the contracts program harms it.

P4283R0: extends contracts on asserted value with no evidence

P4283R0 rests its case for extending the C++26 Contracts facility on an assertion of practical value that it does not support with evidence. The Introduction motivates the feature with worked examples drawn from [P2755R1] and a sample std::vector interface and states that the feature has proven to be very useful in practice, while the Implementation Experience section reports only prototype compiler branches gated behind experimental flags such as -fcontracts-p4283 and cites no production deployment, field data, or user report, none of which exists because the underlying C++26 Contracts facility has not shipped (Introduction; Implementation Experience). Committing committee time to an unproven extension whose value is asserted rather than shown enlarges the contracts program to the detriment of the language.

P4298R0: Anchoring safety-response control in contracts forecloses an independent profiles framework

P4298R0 routes the choice of whether a safety violation may throw through the contracts evaluation-semantic channel, placing control over the safety response inside a substrate that leaves no room for an independent profiles framework.

- The design frames safety configuration as a two-party arrangement between the source-code author and the program builder, leaving no architectural place for a party that imposes a guarantee the build cannot override (Section 3, Section 1).
- Because this configuration reuses the existing resolution pipeline and the P3400R4 labels with no new resolution machinery, and is anchored in the contracts substrate that ships first, an independent framework arrives too late to offer an alternative design for the same setting (Section 4, Section 1, Section 3).
- The paper assigns authority over the runtime response to the builder of the final program and treats source-level intent as overridable at build time, so a programmer's safety guarantee does not survive the build (Section 3, Section 1).
- The new semantics and their termination rule are written into the core language as normative evaluation semantics even though the paper notes the effect is implementable below the language, so altering the behavior later requires a standard revision rather than a library update (Section 2, Section 5, Section 4).
- Semantic selection remains implementation-defined, so the same contract assertion can throw and propagate, continue, or terminate depending on build flags and configuration, and the code cannot be reasoned about in isolation (Section 5, Section 4).

Each of these places the response to a safety violation under the contracts facility and its build-time configuration, confirming that the contracts program advances on ground that leaves no place for an independent profiles framework.

P4298R0 relocates the throw-or-terminate behavior of a contract assertion out of the source and fixes it through out-of-band configuration.

- The same `contract_assert` line propagates an exception in one build and invokes `std::terminate` in another, with the outcome fixed by the `-fcontract-evaluation-semantic=` default, per-group configuration, and JSON configuration files, and nothing at the assertion site indicates which occurs (Implementation Experience, sec. 4; wording sec. 6.11.2 paras 16+a and 17).
- Which evaluation semantic governs a given assertion depends on command-line flags, per-group and JSON configuration, label state, and the installed global violation handler, so code review and static analysis do not establish its behavior from the local source (Implementation Experience, sec. 4; wording sec. 6.11.2).
- The paper adds two special-purpose enumerators, `noexcept_observe` and `noexcept_enforce`, and expands `evaluation_semantic` accordingly, after considering and rejecting a general facet mechanism for the same control (Proposal, sec. 2; Additional Rationale, sec. 3; wording sec. 17.10.1 synopsis and Table 45).
- The behavior is delivered through command-line defaults, per-group configuration, JSON configuration files, a runtime library, and feature-test-gated symbols, expressing it as build-time and runtime configuration infrastructure rather than a self-contained language facility (Implementation Experience, sec. 4).
- The selection is a program-wide build-level decision that separately developed components cannot make independently, and the `nothrow_t` overloads sit in a feature-test-macro-selected inline namespace to avoid an ODR violation across translation units (Additional Rationale, sec. 3; Implementation Experience, sec. 4).

Carrying the exception-safety behavior of an assertion outside its source continues the contracts program's movement of language guarantees into unreviewable configuration, to the detriment of the language.

Conclusion

The papers examined in this analysis are not independent proposals arriving at a committee for evaluation. They are line items in a program, and the program has a direction: route every core-language safety check through the contracts facility, configure that routing through Labels, and deliver the complete infrastructure before profiles exist.

This program harms the language. It puts into the language what a library could provide. It makes continuation past undefined behavior the default and termination the opt-in. It creates semantic variance where the same code means different things under different build flags. It mandates runtime infrastructure that every implementation must carry. These are not incremental extensions of a settled feature. They are departures from the design principles that have governed the language since 1980.

This program harms profiles. It subordinates profiles to the contracts substrate before the committee has decided who owns the response to core-language undefined behavior. It captures the safety configuration design space through Labels. It occupies the class-invariant design space through contracts types. It frames safety as a two-party collaboration between source author and build engineer, leaving no architectural slot for a safety framework that imposes non-overridable guarantees. By the time profiles ship, every assertion in the language will route through the contract-violation handler and be configured by the contracts configuration system. Profiles will arrive not as an independent safety framework but as a naming layer over someone else's machinery.

The architectural question is real. The program-wide contract-violation handler owns the response to core-language undefined behavior under the current design. That question has never been polled by name. It has never been decided on evidence. It has been decided by accretion, one line-item review at a time, and the papers examined here reinforce the framing without re-examining it.

Hardened standard libraries ship today. They deliver terminating runtime checks for array bounds, null pointers, and iterator validity without depending on P2900 contracts, without a program-wide handler, and without Labels. The framework proposed to absorb their configuration exists as a Compiler Explorer prototype. The committee standardizes existing practice. There is no existing practice here.

The committee should not advance these papers without first resolving the ownership question explicitly, on evidence, in a dedicated poll. Profiles must remain a first-class safety framework with their own guarantees, their own enumeration, and their own response. The handler must not own the response to core-language undefined behavior. Field experience must precede architectural consolidation. The ordering matters: specify what has been tried, not what has been prototyped.

Disclosure

The author provides information and serves at the pleasure of the committee.

This paper asks for nothing.

Acknowledgments

References