AGENTIC SEPARATION LOGIC SPECIFICATION SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Specification synthesis, the task of automatically inferring formal specifications from program implementations and natural language, is important for refactoring, transpilation, and optimization, yet remains an open challenge for large software repositories. Existing LLM-based approaches fail to simultaneously scale to such repositories, produce specifications expressive enough to capture systems-code features such as dynamic memory and heap-allocated data structures, and systematically validate those specifications to rule out incorrect candidates. We present Spec-Agent, a model-agnostic inference-time method for synthesizing expressive, well-validated specifications across programming languages and large codebases. Spec-Agent targets a *ladder* of specification languages: propositional logic, first-order logic, propositional separation logic, and first-order separation logic. For each function, Spec-Agent uses static analysis and runtime heap tracing to select the appropriate target specification language, generalizes existing functional tests into fuzz harnesses, and iteratively refines LLM-generated candidates via counterexample-guided feedback. We evaluate the same method on large open-source C++ repositories and a popular open-source Rust codebase. Across the evaluated settings, Spec-Agent synthesizes test-valid specifications for up to **85%** of target functions, with no false positives observed under fuzzing and expert validation in either language, outperforming Claude Code Opus 4.6 at **10**$\times$ lower token cost.

1 INTRODUCTION

A major problem arising from the success of LLM-based automatic programming (Jimenez et al., 2024) is the absence of correctness guarantees for the generated software. Software correctness requires that the code contains no unwanted bugs or security vulnerabilities that could compromise safety (Wang et al., 2025; Fu et al., 2025). More extensive testing exposes previously undetected errors in LLM-generated code (Liu et al., 2023). It is therefore of foremost importance that the next generation of program synthesis agents be equipped with sufficient guardrails. Unfortunately, software verification is undecidable in general (Turing, 1937), with many interesting program properties provably unprovable. While LLMs achieve strong results on program-termination benchmarks, they often fail to produce valid proof witnesses, and their performance declines as program length increases (Sultan et al., 2026).

In this paper, we address the problem of *code contract* synthesis at scale. This is the dual of specification-driven program synthesis: whereas recent systems such as Inductive Deductive Synthesis (Agarwal et al., 2026) generate formally verified software from a supplied specification, we infer specifications that characterize existing, unverified software. Code contracts (Fähndrich, 2010) capture the intent of code without contingent implementation details and serve as formal documentation by characterizing a function's *preconditions* and *postconditions*: they relate inputs to outputs via conditional logic and capture iterative behaviors in logical form. Code contract synthesis enables several important applications: (1) translators can use contracts to migrate legacy code from one language to another (e.g., C/C++ to Rust), and (2) developers can use contracts as unambiguous documentation of function intent.

Prior work has shown the promise of LLMs for automated code contract synthesis (Wen et al., 2024; Ma et al., 2025; Sun et al., 2025; Endres et al., 2024; Ugare et al., 2026), but no existing system

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

meets the demands of real-world systems software in a single pipeline. An ideal contract synthesis system must: (1) scale to repositories spanning millions of lines of code, (2) produce preconditions and postconditions expressive enough to capture rich program features such as dynamic memory and heap-allocated data structures, and (3) systematically validate synthesized candidates to rule out incorrect ones.

Spec-Agent addresses these gaps by combining two techniques that, to our knowledge, have not previously been brought together for specification synthesis. First, we adopt **separation logic** (O'Hearn, 2019), a program logic widely used in verifiers, to express not only the logical intent of a contract but also the resource-aware constraints it imposes on memory, a prerequisite for reasoning about the heap-manipulating code that pervades systems software. Second, we employ **fuzz testing** to drive contract validation: by generalizing each project's existing unit tests into fuzz harnesses, we obtain far broader execution coverage than unit tests alone, and use that coverage to stress-test every candidate contract before accepting it.

While fuzz testing is traditionally a bug-finding technique, we repurpose it as a strong *pseudo-oracle* for specification inference: any input that violates a candidate contract constitutes direct evidence that the contract is wrong, and that input is fed back to the LLM as a counterexample in the next refinement attempt. This is particularly valuable for codebases where full-program verification is unavailable or prohibitively costly (Vanegue & Lahiri, 2013). To our knowledge, Spec-Agent is the first system to couple fuzz testing with specification inference, and the first to do so at the scale of millions of lines of code. We evaluate the same inference method on large C++ repositories and a Rust codebase.

In summary, our work makes three contributions:

- A model-agnostic inference method applicable across programming languages that combines adaptive formal-logic selection, counterexample-guided contract refinement, and automatically generated fuzz harnesses for systematic validation. The method operates on behavioral features and validation feedback shared across languages; we evaluate the same inference procedure on C++ and Rust.
- An adaptive specification-language selector spanning four program logics of increasing expressivity (propositional, first-order, propositional separation, and first-order separation logic), enabling Spec-Agent to reason about conditional behavior, iterative loops, and heap structure within a single pipeline.
- An experimental study spanning large C++ repositories totaling over four million lines of code and a popular open-source Rust codebase, in which Spec-Agent synthesizes test-valid contracts for up to **85%** of target functions with **no** false positives observed under fuzzing and expert validation across both languages, outperforming Claude Code Opus 4.6 at **10**$\times$ lower token cost (Figure 1).

The rest of the paper is organized as follows. Section 2 recalls the program logics and specification languages used by Spec-Agent. Section 3 presents Spec-Agent's design. Section 4 reports our experimental study, including quantitative results and qualitative analysis of the inferred contracts. Section 5 surveys related work, and Section 6 concludes.

2 BACKGROUND

We provide necessary background on program logics and specification languages. Figure 3 shows the formal grammars of the specification languages supported by Spec-Agent.

Program Verification. The gold standard of software correctness is program verification, which guarantees that a program always implements the developer's intent. Pioneered by Floyd (1993) and Hoare (1969), axiomatic verification provides the ultimate criterion for determining whether software satisfies its specification. In Hoare logic, we write $\{P\}\ c\ \{Q\}$ where P is the precondition and Q is the postcondition of code fragment c: any state satisfying P leads to a state satisfying Q after executing c, provided c terminates (partial correctness). The role of a prover such as Frama-C (Cuoq et al., 2012) is to discharge these proof obligations. Creating a specification is thus the first requirement for verification.



Figure 1: Specification validity and cost comparison of Spec-Agent versus Claude Code.

Software Specification. From propositional logic and its familiar operators conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), one can already model simple code contracts, including those with conditional code where guards are encoded as disjunctive implications, e.g., `if` p `then` q_1 `else` q_2 becomes $(p \Rightarrow q_1) \vee (\neg p \Rightarrow q_2)$. Quantifiers are needed for code with loops, where quantifying over objects in a container ensures that they all satisfy desired constraints. Consider the program on the right of Figure 2 that returns `true` if some element of a list satisfies predicate P, or `false` otherwise:

```
pre: x ↦ v1 ⋆ y ↦ v2
swap(int *x, int *y) {
  int z = *x;
  *x = *y;
  *y = z;
}
post: x ↦ v2 ⋆ y ↦ v1
```

```
lookup(std::list<int> &lst) {
  std::list<int>::iterator it =
      lst.begin();
  while (it != lst.end()) {
   if (P(it)) return true; it++;
  }
  return false;
}
```

Figure 2: Left: a memory-manipulating program with separation logic annotations. Right: a function with a loop described with first-order specifications. More examples are given in Appendix E.

The specification for the `lookup` function on the right side of Figure 2 becomes $post: (\forall x \in lst.\ \neg P(x) \Rightarrow ret = \texttt{false}) \vee (\exists x \in lst.\ P(x) \Rightarrow ret = \texttt{true})$, with the precondition being simply `true` since no special conditions on the input are needed for the function to execute correctly.

Separation Logic. Verifying programs with pointers, dynamic memory allocation, and heap-allocated data structures requires separation logic (O'Hearn, 2019), a modular program logic designed for low-level memory-manipulating programs. Separation logic introduces two key constructs: (1) a *maps-to* operator $x \mapsto n$, denoting that memory cell x contains value n, and (2) a *separating conjunction* $\star$, where $p \star q$ asserts that the memory footprints of p and q are strictly disjoint. This is essential for describing heap data structures such as linked lists, trees, and graphs that pervade low-level systems code. It is also needed for simpler examples like `swap` on the left of Figure 2: swapping a variable with itself is ill-defined, and the specification must explicitly state that x and y refer to distinct memory locations. Separation logic is at the core of state-of-the-art program verifier `infer` (Calcagno et al., 2009), enabling compositional reasoning for very large programs, a key enabler for the scalability of our approach.

The predicate-definition production supports named recursive inductive predicates. For example, $\texttt{list}(x) \triangleq (x = \texttt{null}) \vee \exists v, n.\, (x.\texttt{value} \mapsto v) \star (x.\texttt{next} \mapsto n) \star \texttt{list}(n)$ describes an

Logic	Grammar	
Propositional Logic	p :=	$\texttt{true} \mid \texttt{false} \mid p \wedge p \mid p \vee p \mid \neg p$
First Order Logic	fp :=	$p \mid \exists x.fp(x) \mid \forall x.fp(x)$
Propositional Separation Logic	sp :=	$p \mid sp \star sp \mid x \mapsto n$
First Order Separation Logic	fsp :=	$sp \mid \exists x.fsp(x) \mid \forall x.fsp(x) \mid N(\bar{t})$
Inductive Predicate Definition	def :=	$\texttt{PRED}\ N(\bar{x})\ \texttt{:=}\ fsp$

Figure 3: Program specification languages supported by Spec-Agent.

unbounded linked list while making its heap footprint explicit. Predicate bodies may contain recursive calls, points-to atoms, and separating conjunctions. During validation, each definition is interpreted as a recursive executable predicate, bringing inductive specifications into the same counterexample-guided acceptance procedure.

Specification Inference. We address the dual problem of program verification: given a code fragment c, what are the appropriate P and Q for any safe execution? This is the specification inference problem, recovering the weakest precondition (the most permissive condition on inputs) and the strongest postcondition (the most informative condition on outputs given the inputs). Inferring preconditions and postconditions using LLM-assisted tools can enable program translation, full program verification, and serve as formal documentation.

Fuzz Testing. Fuzzing is a technique for dynamically exploring code execution paths (Miller et al., 1990; Zeller et al., 2024). Coverage-guided fuzzing, exemplified by engines such as libFuzzer (Serebryany, 2016) (part of the LLVM toolchain), instruments the target software with a coverage-feedback mechanism and employs a genetic algorithm to discover unique inputs that trigger distinct code paths. Fuzzing is most frequently used for security testing, often combined with sanitizers (Serebryany et al., 2012), to identify inputs that trigger vulnerabilities. LLMs have shown promise for generating fuzz harnesses (Yang et al., 2025); in our work, we leverage coding agents to generate fuzz harnesses for their control-flow exploration capability, using them to stress-test generated contracts far beyond the coverage of unit tests alone.

3 Spec-Agent Design

We formulate Spec-Agent as a general inference method for synthesizing expressive specifications across programming languages and large codebases. Our method is shown in Figure 4 and formalized in Algorithm 1. It maps language-independent observations of control flow, iteration, and heap effects to a target logic, synthesizes and validates a candidate contract, and refines rejected candidates from concrete counterexamples. The target-logic selector, generation objective, lattice acceptance rule, and feedback procedure remain unchanged across programming languages; we evaluate the same method on C++ and Rust.

Behavioral Program Representation. The first stage represents each function f through its source, documentation, existing tests, and behavioral features used by subsequent inference stages. Control-flow observations—including loops, conditionals, and induction-variable usage—are recovered from the program syntax, while heap effects are inferred by executing the existing tests under memory tracing. These observations abstract away surface syntax: iteration signals the need for quantified relations, while heap effects signal the need for ownership and separation constraints.

Fuzz Harness Generation. Fuzz harness generation consists of generalizing existing functional tests into fuzz harnesses. Functional tests target specific hard-coded inputs, while fuzz harnesses are input generators that exercise the function over a wide variety of executions. To produce a harness for function f, Spec-Agent prompts a coding agent with f's signature, source, and existing unit tests. The agent extracts shared test fixtures and setup logic, lifts the hard-coded inputs into fuzzer-controlled parameters, and emits a harness that reuses the original setup while invoking f on fuzzer-generated inputs. This system constitutes a *pseudo-oracle*: it can detect when a candidate specification is wrong (via counterexample), but cannot assert correctness, which would require full formal verification.

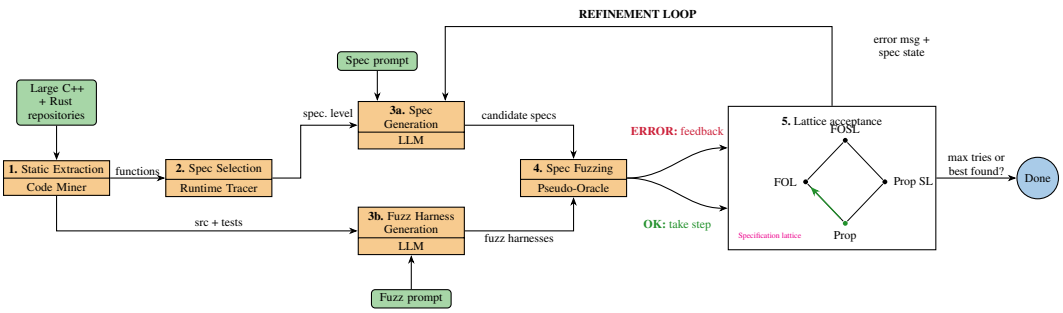


Figure 4: Spec-Agent end to end. Stages 1–2 mine each function and select a target logic L from its static and dynamic features; stages 3a and 3b prompt an LLM for a candidate contract and for a fuzz harness lifted from the project's existing tests; stage 4 stress-tests the assertion-instrumented candidate under fuzz testing; stage 5 accepts only if no counterexample is found *and* the candidate sits at or above L in the specification lattice, otherwise returning feedback to stage 3a. Lattice nodes use the abbreviations of Figure 3.

Algorithm 1 Spec-Agent synthesis, following Figure 4. MINE returns the source src, docs doc and tests tst of function f in repository R; LEVEL picks the target logic L; LIFT generalizes the tests into a fuzz harness H; BUILD compiles candidate $cand$ into the runtime assertion a ($\perp$ if it fails to parse or compile); FUZZ returns a counterexample x ($\perp$ if none). fb is the feedback carried into the next of the N attempts, $best$ the most recent test-valid fallback far, lp and hp whether the function loops and touches the heap, and $\ell(c)$ the lattice level of c. Functions blue, variables purple, constants green.

Algorithm SPECAGENT(f, R, N)
Require: function f, repository R, budget N
Ensure: target-expressive test-valid contract if found; otherwise a test-valid fallback or $\perp$

$src, doc, tst \leftarrow$ MINE(f, R)
$L \leftarrow$ LEVEL(src, tst); $H \leftarrow$ LIFT(src, tst)
$fb \leftarrow \varnothing$; $best \leftarrow \perp$
for $i \leftarrow 1$ **to** N **do**
  $cand \leftarrow$ GEN(src, doc, L, fb)
  $a \leftarrow$ BUILD$(f, cand)$
  if $a = \perp$ **then** $fb \leftarrow$ DIAG$(cand)$; **continue**
  $x \leftarrow$ FUZZ(a, H)
  if $x \neq \perp$ **then** $fb \leftarrow x$; **continue**
  $best \leftarrow cand$
  if $L \sqsubseteq \ell(cand)$ **then return** $cand$
  $fb \leftarrow$ WEAK$(\ell(cand), L)$
end for
return $best$

function LEVEL(src, tst)
  $lp \leftarrow$ LOOP(src); $hp \leftarrow$ HEAP(tst)
  if $lp \wedge hp$ **then return** FOSL
  if hp **then return** Prop SL
  if lp **then return** FOL
  return Prop
end function

Specification Language Selection. Spec-Agent supports four specification languages: propositional logic (Prop), first-order logic (FOL), propositional separation logic (Prop SL), and first-order separation logic (FOSL). Given the behavioral representation, this stage selects a target language L for f. Whenever f contains a loop, first-order logic is required to model predicates over entire collections (e.g., elements of a list or container). Separation logic is selected when f uses dynamic memory allocation. When both heap reasoning and quantification over collections are required, L is set to FOSL.

Specification Generation. We instruct a large language model to generate a candidate specification $\langle P, Q \rangle$ for f based on the selected specification language, the function's source code, and any documentation comments present in the project. Note that unit tests are not used during the specification generation phase, they are reserved for validation. This step is agnostic to the specific LLM used and treats it as a black-box generator. The prompt includes specific instructions for using the chosen specification language, up to 10 manually-written correct examples in that language, and the formal grammar of the selected logic.

Specification Fuzz Testing. Each candidate is parsed against the grammar of L, compiled to an executable runtime assertion, and stress-tested with coverage-guided fuzzing. We use libFuzzer in both evaluated languages, directly for C++ and through `cargo-fuzz`/`libfuzzer-sys` for Rust.

Boolean, arithmetic, and implication operators map to their target-language counterparts. Bounded quantifiers compile to bounded loops over a range $[a, b)$, where the bounds are LLM-supplied expressions over the function's parameters and standard collection or iterator interfaces (e.g., zero and a collection's length, or the endpoints of an iterator range). Separation-logic operators are realized over the observed heap state at runtime: the points-to atom $x \mapsto n$ asserts that the cell at address x holds value n, and the separating conjunction $p \star q$ tracks each side's heap footprint and asserts that the two address sets are disjoint. Separating quantifiers extend $\star$ index-wise across the iteration range. The assertion is exercised by libFuzzer through harnesses lifted from the project's existing unit tests, ensuring the assertion is checked over a broad fuzzer-driven input distribution rather than a handful of hand-written cases. If any input violates the assertion, the candidate is rejected and the failing input together with the violated assertion is recorded as feedback for the next refinement attempt.

Specification Refinement. A candidate is *test-valid* when it parses, compiles, passes the repository tests, and the fuzzer finds no counterexample. It is additionally *target-expressive* when it is expressed in the targeted logic L. The four languages form a partial order $\sqsubseteq$ depicted in Figure 4: Prop $\sqsubseteq$ FOL, Prop $\sqsubseteq$ Prop SL, FOL $\sqsubseteq$ FOSL, and Prop SL $\sqsubseteq$ FOSL, with Prop as the bottom element and FOSL as the top. A candidate sits at the lattice level its operators require, and it is target-expressive if and only if $L \sqsubseteq \ell(cand)$. This condition requires the operator families selected for the target but does not prove that the contract is semantically strongest. The distinction accounts for the heuristic nature of the selector: a function with a loop may not actually need a quantifier in its contract, and a function without an explicit loop may still benefit from a quantified summary. If a test-valid candidate has $\ell(cand) \sqsubset L$ (e.g., it omits $\mapsto$ for a heap-touching function targeted at FOSL), the structural diagnostic is fed back to the LLM for another attempt. The loop returns immediately upon finding a target-expressive candidate; if the retry budget is exhausted, it retains the most recent test-valid candidate as a clearly identified fallback rather than treating lattice level as proof of semantic maximality.

4 Evaluation

Experimental Setup. We evaluate four open-weight models—Qwen3.5-35B-A3B (Qwen Team, 2026), Qwen2.5-32B-Instruct (Qwen Team, 2024), CWM (FAIR CodeGen team et al., 2025), and Qwen3-Coder-Next (Cao et al., 2026)—with greedy decoding and at most 400 tokens per refinement attempt. Experiments run on one Intel Xeon Silver 4410Y CPU and four NVIDIA H100 GPUs using vLLM (Kwon et al., 2023), Clang (LLVM Project, 2026), Tree-sitter (Tree-sitter contributors, 2026), and libFuzzer (via `cargo-fuzz`/`libfuzzer-sys` for Rust). Claude Code Sonnet 4.6 and Opus 4.6 (Anthropic, 2026a) are free-form agentic baselines: they receive the same context, prompts, examples, tests, and fuzz harnesses, and are scored by the same validation criteria. Claude receives up to 6M weighted tokens per repository—more than any Spec-Agent configuration—under the same applicable per-function limits. Costs use published Anthropic and Together AI prices (Anthropic, 2026b; Together AI, 2026).

Following (Ugare et al., 2026), we extract documented, tested public functions from BDE (Bloomberg L.P., 2026a) and `BlazingMQ` (Bloomberg L.P., 2026b) (BMQ), two large C++ repositories, and Tokio, a popular Rust codebase: 651, 508, and 200 functions, respectively. Two separation-logic and systems experts also reviewed every generated fuzz harness and a random sample of 100 specifications per repository, agreeing on all judgments (Appendix F).

Accuracy. We measure test-valid contracts (compiled and not falsified by unit tests or fuzzing), counterexamples, compilation errors, trivial contracts, and mean atom count; test validity is empirical non-falsification, not formal proof. Qwen3-Coder-Next performs best on both C++ repositories: 85.87% test-valid on BDE versus 81.48% for Claude Opus 4.6, and 77.73% on BMQ versus 66.69%. Its contracts are also richer on average (3.35 and 4.15 atoms versus 2.34 and 2.46), and Table 2 shows more test-valid contracts using quantifiers and separation logic. A controlled ablation on 100 heap-manipulating BDE/BMQ functions holds the Qwen3-Coder-Next pipeline and matched examples fixed: separation logic yields 3.7× more test-valid contracts that capture essential memory constraints and reduces mean size from 13 to 4 atoms (Table 3).

Table 1: Model performance with Spec-Agent on large C++ repositories. Bold marks the configuration with the highest Test Valid rate for each repository; diagnostic columns are not optimized independently.

Repo.	Method	Model	Test Valid (%)	Test Invalid (%)	Compile Error (%)	Trivial (%)	Avg. Atoms
BDE	Claude Code	Sonnet 4.6	78.06	4.52	17.42	10.89	2.24
BDE	Claude Code	Opus 4.6	81.48	4.71	13.80	17.34	2.34
BDE	Spec-Agent	CWM	81.00	0.00	19.00	8.30	2.78
BDE	Spec-Agent	Qwen3.5-35B-A3B	85.10	0.00	14.90	10.29	3.21
BDE	Spec-Agent	Qwen2.5-32B	85.85	0.96	13.18	5.63	3.86
BDE	**Spec-Agent**	**Qwen3-Coder-Next**	**85.87**	**0.00**	**14.13**	**5.84**	**3.35**
BMQ	Claude Code	Sonnet 4.6	62.64	2.16	35.19	14.18	2.13
BMQ	Claude Code	Opus 4.6	66.69	2.19	31.12	11.92	2.46
BMQ	Spec-Agent	CWM	69.77	0.00	30.23	8.94	3.07
BMQ	Spec-Agent	Qwen2.5-32B	72.41	0.38	27.20	6.13	3.97
BMQ	Spec-Agent	Qwen3.5-35B-A3B	76.97	0.20	22.84	17.13	4.05
BMQ	**Spec-Agent**	**Qwen3-Coder-Next**	**77.73**	**0.00**	**22.27**	**16.94**	**4.15**

Table 2: Number of test-valid specifications per specification language.

Repo.	Method	Model Name	Prop	FOL	Prop SL	FOSL
BDE	Spec-Agent	Qwen3.5-35B-A3B	289	200	52	13
BDE	Claude Code	Claude Opus 4.6	369	118	42	1
BMQ	Spec-Agent	Qwen3.5-35B-A3B	186	173	26	4
BMQ	Claude Code	Claude Opus 4.6	236	71	29	3

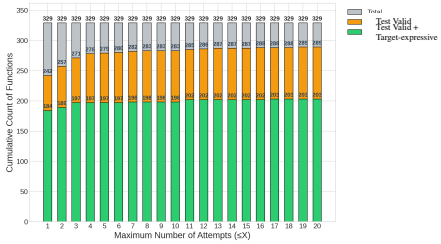
Table 3: Controlled separation-logic ablation on 100 heap-manipulating functions from BDE and BMQ using Qwen3-Coder-Next. The middle column counts contracts that are test-valid and express the essential memory constraints.

Spec-Agent variant	Test-valid + memory	Avg. atoms
Without separation logic	18/100 (18%)	13
With separation logic	**67/100 (67%)**	**4**

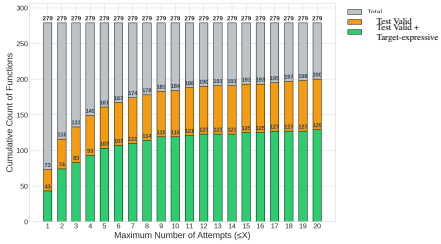
Rust Results. Under the same synthesis, fuzz-validation, and expert-review protocol, Qwen3-Coder-Next reaches 82.0% test-valid on Rust versus 73.0% for Claude Opus 4.6 and lowers compilation errors from 21.0% to 17.5% (Table 4). Appendix B gives per-language results and refinement curves.

Table 4: Model performance with Spec-Agent on the popular open-source Rust codebase. Bold marks the configuration with the highest Test Valid rate; diagnostic columns are not optimized independently.

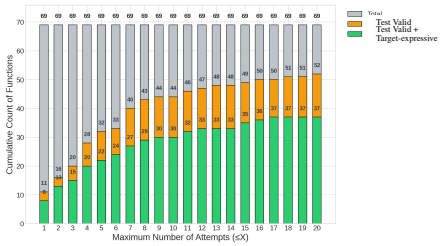
Repo.	Method	Model	Test Valid (%)	Test Invalid (%)	Compile Error (%)	Trivial (%)	Avg. Atoms
Tokio	Claude Code	Sonnet 4.6	66.00	8.00	26.00	–	2.22
Tokio	Claude Code	Opus 4.6	73.00	6.00	21.00	–	2.67
Tokio	Spec-Agent	Qwen3.5-35B-A3B	78.00	0.00	22.00	16.50	3.47
Tokio	**Spec-Agent**	**Qwen3-Coder-Next**	**82.00**	**0.50**	**17.50**	**20.00**	**3.41**



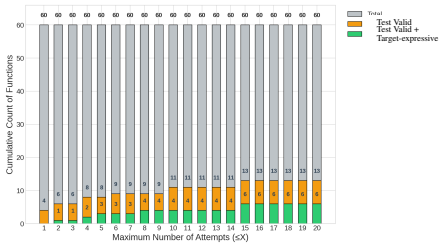
(a) Propositional logic (Prop).



(b) First-order logic (FOL).



(c) Propositional separation logic (Prop SL).



(d) First-order separation logic (FOSL).

Figure 5: Number of BDE functions with test-valid and target-expressive specifications by number of refinement attempts, for each target specification language. Results shown for Spec-Agent with Qwen3.5-35B-A3B.

Refinement and Failures. Figure 5 shows rapid gains in the first five retries and diminishing returns by 20. Unlike Claude Code's open-ended subagent exploration, Spec-Agent focuses each retry on validator feedback for a preselected logic. The green bars satisfy $L \sqsubseteq \ell(cand)$, a minimum operator-level requirement rather than semantic maximality. Most Prop, FOL, and Prop SL targets reach this level, whereas FOSL plateaus. A representative failure is copying a cyclic, shared heap graph: models may preserve values but omit edges, aliasing, freshness, disjointness, or framing, and often fail to assemble the complete quantified recursive relation within the retry budget.

Table 5: Fuzz testing results: *Passed* = no counterexample, *Violated* = counterexample found, *Timeout* = fuzz binary timed out.

Dataset	Passed	Violated	Timeout
BDE	98.3%	0.4%	1.2%
BMQ	92.0%	8.0%	0.0%

Impact of Fuzzing on Contract Quality. After candidates pass repository unit tests, fuzzing still rejects 0.4% of BDE and 8.0% of BMQ survivors (Table 5), finding counterexamples missed by the fixed tests.

Table 6: Comparison with prior work. ✓ denotes the work has this capability, and ✗ denotes that the capability is not targeted. "Test Gen. / Verification" denotes either dynamically generated or fuzzed tests beyond fixed unit tests, or verifier-based validation of synthesized specifications.

Work	Large C++ Projects	Pre + Post	Separation Logic	Adaptive Spec Lang. Selection	Test Gen. / Verification	Counterex. Repair
Daikon	✗	✓	✗	✗	✗	✗
SLING	✗	✓	✓	✗	✗	✗
NL2Postcond	✗	✗	✗	✗	✗	✗
LLM-SE	✗	✗	✓	✗	✓	✗
AutoSpec	✗	✓	✗	✗	✓	✗
ClassInvGen	✗	✓	✗	✗	✓	✓
SpecGen	✗	✓	✗	✗	✓	✗
Fun2Spec	✓	✗	✗	✗	✗	✓
Spec-Agent (our work)	✓	✓	✓	✓	✓	✓

5 Related Work

Dynamic approaches include Daikon (Ernst et al., 2007), which infers likely contracts from traces, and SLING (Le et al., 2019), which extends trace-based inference to separation logic. LLM methods use documentation and validator feedback: AutoSpec (Wen et al., 2024) targets C with Frama-C; SpecGen (Ma et al., 2025) targets Java with OpenJML; ClassInvGen (Sun et al., 2025) infers C++ class invariants; LLM-SE (Liu et al., 2024) generates separation-logic loop invariants; and NL2Postcond (Endres et al., 2024) translates docstrings into postconditions. Most target small benchmarks. Fun2Spec (Ugare et al., 2026) scales to large C++ repositories but generates only postconditions, omits separation logic, and validates only against fixed tests. Table 6 shows that Spec-Agent uniquely combines repository scale across C++ and Rust with pre- and postconditions, separation logic, adaptive language selection, generated tests, and counterexample repair.

6 Conclusion

Spec-Agent is a model-agnostic inference-time method for specification synthesis across programming languages. Its formal-logic selection, validator-guided refinement, and acceptance criterion apply unchanged across languages; results on large C++ and Rust codebases demonstrate that this general method produces more test-valid specifications than Claude Code at substantially lower cost. Separation logic improves coverage of essential memory constraints and formula compactness, while the FOSL plateau identifies a concrete capability frontier for future inference methods.

AI USE STATEMENT

We used AI tools for manuscript editing and LaTeX formatting. All AI-generated outputs were reviewed by the authors and edited for accuracy and clarity. The authors take full responsibility for the final manuscript, implementation, and reported results.

ETHICS STATEMENT

This work develops an automated system for synthesizing and validating code contracts across large software codebases. Our experiments use previously released open-source repositories and their existing tests. Improved specification synthesis may support documentation, verification, translation, and vulnerability discovery, but inaccurate contracts or automated security analysis could cause harm if treated as proofs or used without appropriate oversight. The reported validation relies on fuzzing and expert review rather than full program verification. Real-world use should preserve repository licenses, avoid exposing sensitive code, sandbox generated harnesses, review synthesized contracts, and conduct separate assessments of security, privacy, and misuse risks.

REPRODUCIBILITY STATEMENT

Section 3 and Algorithm 1 describe the Spec-Agent pipeline, specification-language selection, fuzz-harness generation, and counterexample-guided refinement. Sections 4, 4, and 4 describe the repositories, models, experimental setup, evaluation metrics, and fuzz-based validation. Appendix A reports additional runtime, coverage, and function-size distributions; Appendix B reports the per-language Rust results and refinement curves; Appendix D details the cost calculation; and Appendix E provides end-to-end inferred contract examples.

REFERENCES

Shubham Agarwal, Alexander Krentsel, Shu Liu, Mert Cemri, Audrey Cheng, Rui Meng, Tomas Pfister, Chun-Liang Li, Sylvia Ratnasamy, Aditya Parameswaran, Matei Zaharia, Ion Stoica, and Mohsen Lesani. Inductive deductive synthesis: Enabling AI to generate formally verified systems, 2026. URL https://arxiv.org/abs/2605.23109.

Anthropic. Claude Code: Overview. Software documentation, 2026a. URL https://code.claude.com/docs/en/overview. Accessed: 2026-09-26.

Anthropic. Claude platform pricing. API documentation, 2026b. URL https://platform.claude.com/docs/en/about-claude/pricing. Accessed: 2026-09-26.

Bloomberg L.P. BDE libraries. Software repository, 2026a. URL https://github.com/bloomberg/bde. Accessed: 2026-09-26.

Bloomberg L.P. BlazingMQ: A modern, high-performance message queue. Software repository, 2026b. URL https://github.com/bloomberg/blazingmq. Accessed: 2026-09-26.

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 289–300, 2009.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-Coder-Next technical report. *arXiv preprint arXiv:2603.00729*, 2026. URL https://arxiv.org/abs/2603.00729.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *International conference on software engineering and formal methods*, pp. 233–247. Springer, 2012.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE):1889–1912, 2024. doi: 10.1145/3660791. URL https://doi.org/10.1145/3660791.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. doi: 10.1016/j.scico.2007.01.015. URL https://doi.org/10.1016/j.scico.2007.01.015.

Manuel Fähndrich. Static verification for code contracts. In *International Static Analysis Symposium*, pp. 2–5. Springer, 2010.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. CWM: An open-weights LLM for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

Robert W Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pp. 65–81. Springer, 1993.

Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. Security weaknesses of Copilot-generated code in GitHub projects: An empirical study, 2025. URL https://arxiv.org/abs/2310.02059.

Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In Kathryn S. McKinley and Kathleen Fisher (eds.), *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pp. 788–801. ACM, 2019. doi: 10.1145/3314221.3314634. URL https://doi.org/10.1145/3314221.3314634.

Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. Towards general loop invariant generation: A benchmark of programs with memory manipulation. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/e93b673c55d6768cdd39ce90de8c4d4c-Abstract-Datasets_and_Benchmarks_Track.html.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

LLVM Project. Clang: A C language family frontend for LLVM. Software documentation, 2026. URL https://clang.llvm.org/docs/. Accessed: 2026-09-26.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pp. 16–28. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00129. URL https://doi.org/10.1109/ICSE55347.2025.00129.

Barton Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33:32–44, 12 1990. doi: 10.1145/96267.96279.

Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh (eds.), *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, pp. 309–318. USENIX Association, 2012. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

Kosta Serebryany. Continuous fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pp. 157–157, 2016. doi: 10.1109/SecDev.2016.043.

Oren Sultan, Jordi Armengol-Estape, Pascal Kesseli, Julien Vanegue, Dafna Shahaf, Yossi Adi, and Peter O'Hearn. LLMs versus the halting problem: Revisiting program termination prediction. *arXiv preprint arXiv:2601.18987v1*, 2026. URL https://arxiv.org/abs/2601.18987v1.

Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark W. Barrett, David L. Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. ClassInvGen: Class invariant synthesis using large language models. In Mirco Giacobbe and Anna Lukina (eds.), *AI Verification - Second International Symposium, SAIV 2025, Zagreb, Croatia, July 21-22, 2025, Proceedings*, Lecture Notes in Computer Science, pp. 64–96. Springer, 2025. doi: 10.1007/978-3-031-99991-8_4. URL https://doi.org/10.1007/978-3-031-99991-8_4.

Together AI. Together ai pricing, 2026. URL https://www.together.ai/pricing.

Tree-sitter contributors. Tree-sitter: An incremental parsing system for programming tools. Software repository, 2026. URL https://github.com/tree-sitter/tree-sitter. Accessed: 2026-09-26.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: 10.1112/plms/s2-42.1.230.

Shubham Ugare, Tarun Suresh, Sasa Misailovic, and Julien Vanegue. Fun2Spec: Code contract synthesis at scale. In *Companion Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering (FSE Companion '26), Industry Track*, Montreal, QC, Canada, 2026. ACM. URL https://conf.researchr.org/details/fse-2026/fse-2026-industry-papers/60/Fun2spec-Code-Contract-Synthesis-At-Scale.

Julien Vanegue and Shuvendu K Lahiri. Towards practical reactive security audit using extended static checkers. In *2013 IEEE Symposium on Security and Privacy*, pp. 33–47. IEEE, 2013.

Xiaoqing Wang, Keman Huang, Bin Liang, Hongyu Li, and Xiaoyong Du. Shadows in the code: Exploring the risks and defenses of LLM-based multi-agent software development systems, 2025. URL https://arxiv.org/abs/2511.18467.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In Arie Gurfinkel and Vijay Ganesh (eds.), *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, Lecture Notes in Computer Science, pp. 302–328. Springer, 2024. doi: 10.1007/978-3-031-65630-9_16. URL https://doi.org/10.1007/978-3-031-65630-9_16.

Kang Yang, Yunhang Zhang, Zichuan Li, Guanhong Tao, Jun Xu, and Xiaojing Liao. HarnessAgent: Scaling automatic fuzzing harness construction with tool-augmented LLM pipelines. *ArXiv*, abs/2512.03420, 2025. URL https://arxiv.org/abs/2512.03420.

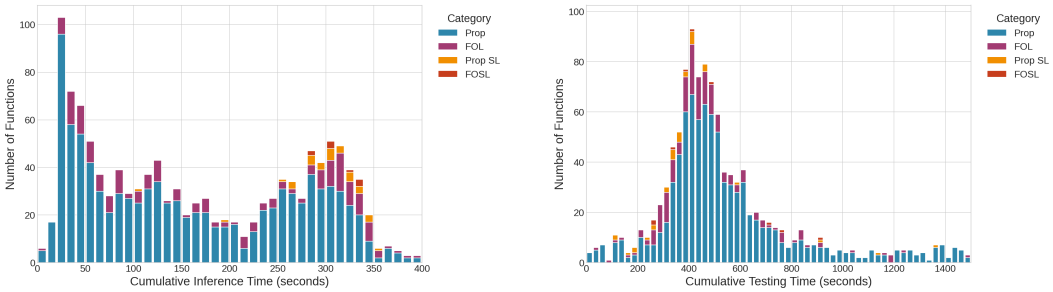
Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL https://www.fuzzingbook.org/.

A Additional Repository and Fuzzing Statistics

This appendix provides runtime, coverage, and function-size distributions that supplement the main evaluation.

A.1 Cumulative Inference and Testing Time

Figure 6 shows the runtime distribution of contract inference (via LLM) and contract testing (via fuzz testing). The choice of target logic does not noticeably influence runtime, as all specification languages are evenly distributed across the timing spectrum. Additionally, the overall runtime is dominated by testing rather than inference, with cumulative per-function means of 170 seconds for inference and 400 seconds for testing.



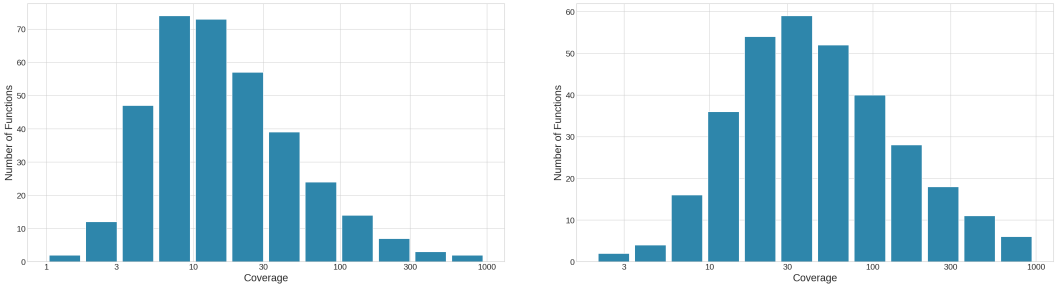
(a) Cumulative inference time.

(b) Cumulative testing time.

Figure 6: Distribution of cumulative inference and testing time for each BDE function with an accepted synthesized specification, for Spec-Agent with Qwen3.5-35B-A3B.

A.2 Fuzz-harness Coverage and Function Size

The following distributions show the per-harness branch coverage of the generated fuzz harnesses. The X axis is given in logarithmic scale while the Y axis is given in linear scale.



(a) BDE.

(b) BMQ.

Figure 7: Coverage distributions for Spec-Agent's LLM-generated fuzz harnesses on BDE and BMQ.

We can witness that our fuzz testing harnesses achieve strong coverage for both projects, the number of reached basic blocks being normally distributed.

We also report the distribution of lines of code per function across the evaluated functions in BDE and BMQ in Figure 8. Both repositories exhibit a similar long-tailed distribution: most functions are short (5–14 lines), while a non-trivial fraction span 20 or more lines, with a small number of functions exceeding 100 lines.

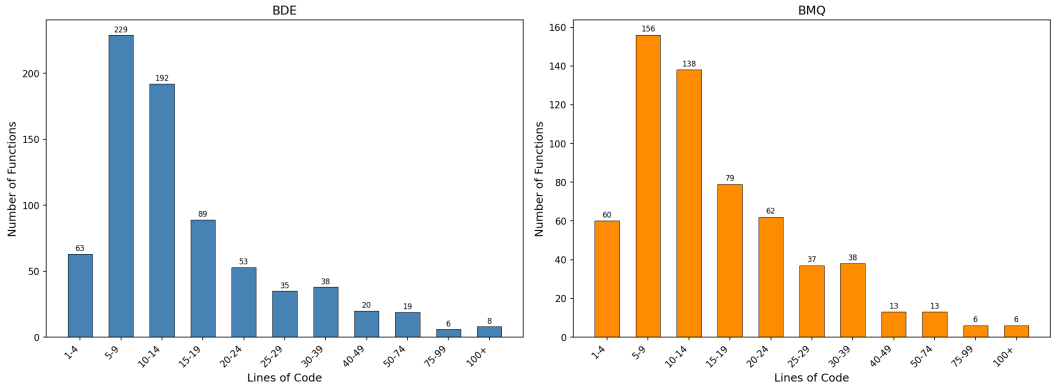


Figure 8: Distribution of lines of code per function for the evaluated functions in BDE (left) and BMQ (right).

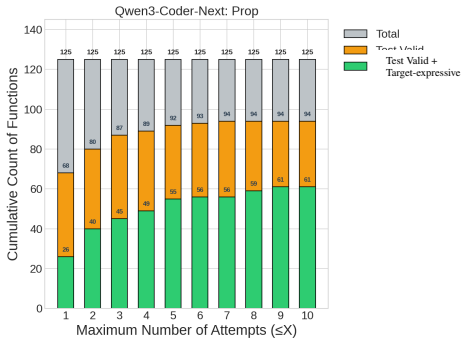
B Rust Results by Specification Language

Table 7 reports the number of test-valid Rust specifications synthesized in each target language. Qwen3-Coder-Next produces more separation-logic specifications than Claude Code Opus 4.6: 58 versus 25 for Prop SL and 8 versus 4 for FOSL.

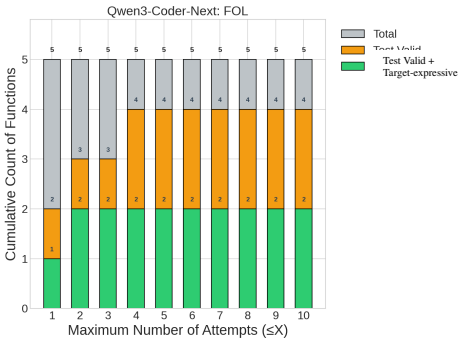
Table 7: Number of test-valid specifications on the popular open-source Rust codebase.

Repo.	Method	Model Name	Prop	FOL	Prop SL	FOSL
Rust codebase	Claude Code	Sonnet 4.6	104	4	22	2
Rust codebase	Claude Code	Opus 4.6	110	7	25	4
Rust codebase	Spec-Agent	Qwen3.5-35B-A3B	112	7	33	4
Rust codebase	Spec-Agent	Qwen3-Coder-Next	94	4	58	8

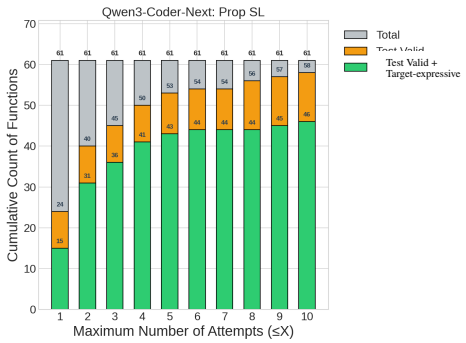
Figure 9 shows the cumulative Qwen3-Coder-Next results as the maximum number of refinement attempts increases. Gains are largest in the first few attempts and then taper, with the test-valid totals reaching 145 for Prop, 146 for FOL, 93 for Prop SL, and 86 for FOSL by ten attempts.



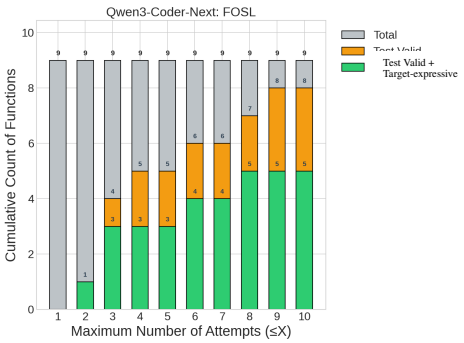
(a) Propositional logic (Prop).



(b) First-order logic (FOL).



(c) Propositional separation logic (Prop SL).



(d) First-order separation logic (FOSL).

Figure 9: Number of test-valid and target-expressive Rust specifications synthesized by Qwen3-Coder-Next as the maximum number of refinement attempts increases.

C LIMITATIONS

Fuzz Testing vs. Full Verification. While our specifications are broadly fuzz-tested with high execution coverage and thousands of test inputs per specification (generalizing from narrow functional tests), Spec-Agent does not establish full correctness using a deductive verifier. Existing contract verifiers do not cover the full programming languages and repository-scale systems studied here, and fuzz testing constitutes a strong, scalable alternative that supersedes functional testing, the typical approach used to validate inferred specifications in prior work. Developing a sound verifier for these complete languages and codebases is out of scope for this paper.

Target Expressivity vs. Semantic Completeness. The lattice check establishes that a candidate uses the operator families selected for the function; it does not prove that the contract captures every possible program behavior. A target-expressive contract may still omit a relevant relation or describe only the executions exercised by the validation distribution. Consequently, our claims concern empirical test validity and minimum target expressivity, not semantic completeness or formal correctness. This distinction is most important for FOSL functions, where quantified heap relationships, aliasing, and frame conditions remain difficult for current models to synthesize within the retry budget.

D COST BREAKDOWN ON BDE

Table 8: Per-model token usage and total dollar cost on BDE. Token counts are in millions of tokens (Mtok).

Method	Model	Input (Mtok)	Output (Mtok)	Total Cost ($)
Spec-Agent	Qwen2.5-32B	3.0	0.7	3.5
Spec-Agent	Qwen3-Coder-Next	3.2	1.0	4.0
Spec-Agent	Qwen3.5-35B-A3B	2.6	0.6	3.0
Spec-Agent	CWM	3.5	1.2	5.5
Claude Code	Sonnet 4.6	3.1	2.0	24.8
Claude Code	Opus 4.6	3.3	2.2	41.0

To accompany the cost-versus-accuracy comparison in Figure 1, Table 8 reports the full breakdown of input tokens, output tokens, per-token pricing, and total dollar spend for every model we evaluate, on BDE. For Spec-Agent, the reported tokens sum across every LLM call made by the system, fuzz harness generation, specification generation, and specification refinement. For Claude Code, the reported tokens aggregate over all top-level and subagent calls performed within the per-repository budget.

E END-TO-END INFERRED CODE CONTRACTS ON LARGE C++ PROJECTS

We illustrate two complete examples from BDE and BMQ, for which we infer pre and post conditions. We use the following notations throughout the contracts to encode specifications:

$$
\begin{aligned}
\texttt{FORALL}(a, i, b, P) &\Leftrightarrow \bigwedge_{i=a}^{b} P(i) \\
\texttt{SEPFORALL}(a, i, b, P) &\Leftrightarrow \mathop{\ast}_{i=a}^{b} P(i) \\
\texttt{EXISTS}(a, i, b, P) &\Leftrightarrow \exists i.a \leq i \wedge i < b \wedge P(i) \text{ when } \star \text{ or } \mapsto \text{ may not be in } P \\
\texttt{SEPEXISTS}(a, i, b, P) &\Leftrightarrow \exists i.a \leq i \wedge i < b \wedge P(i) \text{ when } \star \text{ or } \mapsto \text{ may be in } P
\end{aligned}
$$

```
// requires: true
// ensures:
// (__out == true ==>
//  EXIST(trans.begin(), trans.end(), it, descr == it->descr())) &&
// (__out == false ==>
// FORALL(trans.begin(), trans.end(), it, descr != it->descr()))
static bool containsDescriptor(
const bsl::vector<baltzo::ZoneinfoTransition>& trans,
const baltzo::LocalTimeDescriptor& descr) {
    auto it  = trans.begin();
    auto end = trans.end();
    for (; it != end; ++it)
        if (descr == it->descriptor())
            return true;
    return false;
}
```

Figure 10: First-order logic contract capturing a loop invariant directly in the postcondition (ensures), allowing any input in the precondition (requires).

```
// requires:
// SEP_FORALL(0, rhs.d_len, i, (rhs.d_head_p + i)->d_value_p ↦ _)
// ensures:
// (__out == *this) && (__out.d_length == rhs.d_length) &&
// SEP_FORALL(0, rhs.d_length, i,
//       __out.d_head_p->d_value_p + i ↦ rhs.d_head_p->d_value_p + i)
AttributeContainerList&
AttributeContainerList::operator=(const AttributeContainerList& rhs) {
    if (&rhs != this) {
        // Append the 'rhs' elements to this list.
        Node **prevNextAddr = &d_head_p;
        node = new (*d_allocator.mechanism()) Node();
        for (iterator it = rhs.begin(); it != rhs.end(); ++it) {
            node->d_value_p = *it;
            node->d_next_p = 0;
            node->d_prevNextAddr_p = prevNextAddr;
            *prevNextAddr = node;
            prevNextAddr = &node->d_next_p;
            ++d_length;
        }
    }
    return *this;
}
```

Figure 11: A first-order separation logic contract capturing deep memory constraints between C++ class attributes using the separation logic *points-to* ($\mapsto$) relation and separating conjunctions between subformulae (implicit in SEP_FORALL), guaranteeing non-aliasing of memory cells.

F CASE STUDY

We list 100 examples of rich C++ specifications synthesized by Spec-Agent on `BDE` (Bloomberg L.P., 2026a) and `BlazingMQ` (Bloomberg L.P., 2026b). The specifications are expressed in various program logics, including propositional logic (red box), first-order logic (yellow box), separation logic (green box), and first-order separation logic (blue box). The specification language is chosen as described in section 3 paragraph 3.

1. **Filename:** balcl_optiontype.cpp, **Function:** OptionType::print, **Logic: First Order Separation Logic**, [code]

   requires: true
   ensures: __out == stream && (SEPFORALL(0, toAscii(value).size(), i, stream + i ↦ toAscii(value)[i]))

2. **Filename:** baljsn_encoder_testtypes.cpp, **Function:** EncoderTestChoiceWithAllCategoriesEnumeration::fromString, **Logic: First Order Separation Logic**, [code]

   requires: result != 0 && string != 0 && stringLength >= 0
   ensures: (__out == 0 ==> SEPEXISTS(0, 2, i, (stringLength == EncoderTestChoiceWithAllCategoriesEnumeration::ENUMERATOR_INFO_ARRAY[i].d_nameLength && 0 == bsl::memcmp(EncoderTestChoiceWithAllCategoriesEnumeration::ENUMERATOR_INFO_ARRAY[i].d_name_p, string, stringLength) && (*result == static_cast<EncoderTestChoiceWithAllCategoriesEnumeration::Value>(EncoderTestChoiceWithAllCategoriesEnumeration::ENUMERATOR_INFO_ARRAY[i].d_value))))) && (__out == -1 ==> SEPFORALL(0, 2, i, !(stringLength == EncoderTestChoiceWithAllCategoriesEnumeration::ENUMERATOR_INFO_ARRAY[i].d_nameLength && 0 == bsl::memcmp(EncoderTestChoiceWithAllCategoriesEnumeration::ENUMERATOR_INFO_ARRAY[i].d_name_p, string, stringLength))))

3. **Filename:** ball_attributecontainerlist.cpp, **Function:** AttributeContainerList::operator=, **Logic: First Order Separation Logic**, [code]

   requires: SEPFORALL(rhs.begin(), rhs.end(), it, *it ↦ _)
   ensures: __out == *this && (__out == *this ==> SEPFORALL(rhs.begin(), rhs.end(), it, EXISTS(__out.begin(), __out.end(), jt, *jt == *it)))

4. **Filename:** ball_log.cpp, **Function:** Log::format, **Logic: First Order Separation Logic**, [code]

   requires: format != 0 && buffer != 0 && SEPFORALL(0, numBytes, i, buffer + i ↦ _)
   ensures: (__out == -1) || (__out != -1 && SEPFORALL(0, __out, i, buffer + i ↦ _))

5. **Filename:** ball_userfields.cpp, **Function:** UserFields::print, **Logic: First Order Separation Logic**, [code]

   requires: true
   ensures: (stream.bad() ==> __out == stream) && (stream.good() ==> SEPFORALL(0, length(), i, EXISTS(0, __out.size(), j, (__out + j) ↦ value(i))))

6. **Filename:** balm_metricdescription.cpp, **Function:** MetricDescription::print, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out == stream && (SEPFORALL(0, d_category_p->name().size(), i, stream + i ↦ d_category_p->name()[i]) ⋆ (stream + d_category_p->name().size() ↦ '.') ⋆ SEPFORALL(0, d_name_p.size(), j, stream + d_category_p->name().size() + 1 + j ↦ d_name_p[j]))

7. **Filename:** balm_metricid.cpp, **Function:** MetricId::print, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: (d_description_p == 0 ==> SEPFORALL(0, strlen("INVALID_ID"), i, stream + i ↦ "INVALID_ID"[i])) && (d_description_p != 0 ==> SEPFORALL(0, strlen(*d_description_p), i, stream + i ↦ (*d_description_p)[i])) && (__out == stream)

8. **Filename:** balm_metricsample.cpp, **Function:** MetricSample::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good() && (spacesPerLevel >= 0)
ensures: __out == stream && (stream.bad() || (stream.good() && SEPFORALL(0, strlen(NL), i, stream + i ↦ NL[i])))

9. **Filename:** balst_stacktraceprinter.cpp, **Function:** balst::operator<<, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out == stream && (stream + 0 ↦ bsl::endl ⋆ SEPFORALL(0, balst::StackTracePrintUtil::printStackTrace(stream, object.d_maxFrames, object.d_demanglingPreferredFlag, object.d_additionalIgnoreFrames + 1) - 1, i, stream + i + 1 ↦ sep_v))

10. **Filename:** balst_stacktracetestallocator.cpp, **Function:** StackTraceTestAllocator::allocate, **Logic: First Order Separation Logic**, [code]

requires: size >= 0
ensures: (size == 0 && __out == 0) || (size != 0 && __out != 0 && (__out ↦ _ ⋆ (__out - 1) ↦ _ ⋆ SEPFORALL(0, d_traceBufferLength, i, (__out - 1 - i) ↦ _)))

11. **Filename:** balst_stacktraceutil.cpp, **Function:** findBasename, **Logic: Propositional Logic**, [code]

requires: pathName != nullptr && strlen(pathName) > 0
ensures: (__out >= pathName) && (__out == pathName || (*(__out - 1) == '/' || *(__out - 1) == '\\'))

12. **Filename:** baltzo_zoneinfo.cpp, **Function:** containsDescriptor, **Logic: First Order Separation Logic**, [code]

requires: (res_tmp == true ==> SEPEXISTS(0, transitions.size(), i, transitions[i].descriptor() == descriptor)) && (res_tmp == false ==> SEPFORALL(0, transitions.size(), i, transitions[i].descriptor() != descriptor))
ensures: (__out == true ==> SEPEXISTS(0, transitions.size(), i, transitions[i].

descriptor() == descriptor)) && (__out == false ==> SEPFORALL(0, transitions.size(), i, transitions[i].descriptor() != descriptor))

13. **Filename:** baltzo_zoneinfobinaryreader.cpp, **Function:** areAllPrintable, **Logic: First Order Separation Logic**, [code]

    requires: buffer != 0 && length >= 0 && SEPFORALL(0, length, i, (buffer + i ↦ _))
    ensures: (__out == true ==> SEPFORALL(0, length, i, (buffer + i ↦ sep_v) && bdlb::CharType::isPrint(sep_v))) && (__out == false ==> SEPEXISTS(0, length, i, (buffer + i ↦ sep_v) && !bdlb::CharType::isPrint(sep_v)))

14. **Filename:** balxml_prefixstack.cpp, **Function:** PrefixStack::lookupNamespaceId, **Logic: First Order Separation Logic**, [code]

    requires: true
    ensures: (SEPEXISTS(0, d_numPrefixes, i, (d_prefixes[i].first == prefix) && (__out == d_prefixes[i].second))) || (SEPFORALL(0, d_numPrefixes, i, (d_prefixes[i].first != prefix)) && (__out == lookupPredefinedPrefix(prefix).d_nsid))

15. **Filename:** balxml_typesparserutil.cpp, **Function:** parseUnsignedInt, **Logic: First Order Separation Logic**, [code]

    requires: inputLength > 0 && SEPFORALL(0, inputLength, i, (input + i ↦ _ && input[i] >= '0' && input[i] <= '9'))
    ensures: (__out == BAEXML_SUCCESS) || (__out == BAEXML_FAILURE)

16. **Filename:** balxml_typesprintutil.cpp, **Function:** printTextReplacingXMLEscapes, **Logic: First Order Separation Logic**, [code]

    requires: (dataLength >= 0 && SEPFORALL(0, dataLength, i, (data + i) ↦ _)) || (dataLength == -1 && SEPEXISTS(0, strlen(data), i, (data + i) ↦ _)) && stream.good()
    ensures: (__out == 0 ==> stream.good()) && (__out != 0 ==> stream.bad())

17. **Filename:** bbldc_daycountconvention.cpp, **Function:** DayCountConvention::print, **Logic: First Order Separation Logic**, [code]

    requires: true
    ensures: __out == stream && (SEPFORALL(0, strlen(toAscii(value)), i, (stream + i) ↦ toAscii(value)[i]))

18. **Filename:** bdlb_bitstringutil.cpp, **Function:** indent, **Logic: First Order Separation Logic**, [code]

    requires: level >= 0 && spacesPerLevel != 0
    ensures: __out == stream && SEPFORALL(0, level * spacesPerLevel, i, (stream + i ↦ ' '))

19. **Filename:** bdlb_string.cpp, **Function:** String::areEqualCaseless, **Logic: First Order Separation Logic**, [code]

requires: lhsString != nullptr && rhsString != nullptr && SEPFORALL(0, strlen(lhsString), i, (lhsString + i ↦ _)) && SEPFORALL(0, strlen(rhsString), i, (rhsString + i ↦ _))
ensures: (__out == true ==> SEPFORALL(0, strlen(lhsString), i, (bdlb::CharType::toLower(lhsString[i]) == bdlb::CharType::toLower(rhsString[i])))) && (__out == false ==> SEPEXISTS(0, strlen(lhsString), i, (bdlb::CharType::toLower(lhsString[i]) != bdlb::CharType::toLower(rhsString[i]))) || strlen(lhsString) != strlen(rhsString))

20. **Filename:** bdlbb_blob.cpp, **Function:** BlobBuffer::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good()
ensures: __out == stream && (SEPFORALL(0, d_size, i, stream + i ↦ d_buffer.get()[i]) ⋆ stream.flushed())

21. **Filename:** bdlc_indexclerk.cpp, **Function:** IndexClerk::isInUse, **Logic: First Order Separation Logic**, [code]

requires: (unsigned int)index < (unsigned int)d_nextNewIndex
ensures: (__out == false ==> SEPEXISTS(0, d_unusedStack.size(), i, d_unusedStack[i] == index)) && (__out == true ==> SEPFORALL(0, d_unusedStack.size(), i, d_unusedStack[i] != index))

22. **Filename:** bdld_datum.cpp, **Function:** DatumArrayRef::print, **Logic: First Order Separation Logic**, [code]

requires: stream.bad() || stream.good()
ensures: __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, d_length, i, stream + i ↦ d_data_p[i])) && stream.flushed()))

23. **Filename:** bdld_datum.cpp, **Function:** DatumMapEntry::print, **Logic: First Order Separation Logic**, [code]

requires: !stream.bad()
ensures: __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, strlen(d_key_p), i, stream + i ↦ d_key_p[i]) ⋆ SEPFORALL(0, strlen(d_value), j, stream + strlen(d_key_p) + 1 + j ↦ d_value[j]))))

24. **Filename:** bdld_datumbinaryref.cpp, **Function:** DatumBinaryRef::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good()
ensures: (stream.bad() ==> __out == stream) && (stream.good() ==> (__out == (stream << bsl::flush) && (SEPFORALL(0, d_data_p.size(), i, stream + i ↦ d_data_p[i]) ⋆ (stream + d_data_p.size() ↦ ' ') ⋆ SEPFORALL(0, d_size.size(), j, stream + d_data_p.size() + 1 + j ↦ d_size[j]))))

25. **Filename:** bdld_datummaker.cpp, **Function:** DatumMaker::operator(), **Logic: First Order Separation Logic**, [code]

> **requires:** SEPFORALL(0, size, i, elements + i ↦ _) && size >= 0
> **ensures:** __out == bdld::Datum::adoptMap(map) && (SEPFORALL(0, size, i, map.data()[i] == elements[i]) ⋆ (*map.size() == size) ⋆ (*map.sorted() == sorted))

26. **Filename:** bdlde_base64alphabet.cpp, **Function:** Base64Alphabet::print, **Logic: First Order Separation Logic**, [code]

> **requires:** !stream.bad() && (value >= Base64Alphabet::Enum::MIN && value <= Base64Alphabet::Enum::MAX)
> **ensures:** __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, strlen(Base64Alphabet::toAscii(value)), i, (stream + i) ↦ Base64Alphabet::toAscii(value)[i]))))

27. **Filename:** bdlde_base64decoderoptions.cpp, **Function:** Base64DecoderOptions::print, **Logic: First Order Separation Logic**, [code]

> **requires:** stream.good()
> **ensures:** __out == stream && (SEPFORALL(0, 3, i, (__out + i ↦ sep_v && sep_v == printer_attribute[i])))

28. **Filename:** bdlde_base64ignoremode.cpp, **Function:** Base64IgnoreMode::print, **Logic: First Order Separation Logic**, [code]

> **requires:** stream.good()
> **ensures:** __out == stream && (stream.bad() || (stream.good() && SEPFORALL(0, strlen(Base64IgnoreMode::toAscii(value)), i, (stream + i ↦ Base64IgnoreMode::toAscii(value)[i]))))

29. **Filename:** bdlde_charconvertstatus.cpp, **Function:** CharConvertStatus::print, **Logic: First Order Separation Logic**, [code]

> **requires:** !stream.bad()
> **ensures:** __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, strlen(CharConvertStatus::toAscii(value)), i, stream + i ↦ CharConvertStatus::toAscii(value)[i]))))

30. **Filename:** bdlde_charconvertutf32.cpp, **Function:** decodeTwoOctets, **Logic: First Order Separation Logic**, [code]

> **requires:** SEPFORALL(0, 2, i, octBuf + i ↦ _)
> **ensures:** __out == ((octBuf[1] & ~k_CONTINUE_MASK) | ((octBuf[0] & ~k_TWO_OCTET_MASK) << k_CONTINUE_CONT_WID))

31. **Filename:** balm_metricformat.cpp, **Function:** MetricFormatSpec::formatValue, **Logic: First Order Separation Logic**, [code]

> **requires:** true
> **ensures:** __out == stream && (SEPFORALL(0, strlen(buffer), i, stream + i ↦ buffer[i]) || SEPFORALL(0, newBuffer.size(), i, stream + i ↦ newBuffer.data()[i]))

32. **Filename:** bdljsn_error.cpp, **Function:** bdljsn::operator<<, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out == stream && (SEPFORALL(0, object.location().size(), i, stream + i ↦ object.location().data()[i]) ⋆ SEPFORALL(0, object.message().size(), j, stream + object.location().size() + j ↦ object.message().data()[j]))

33. **Filename:** bdljsn_jsontestsuiteutil.cpp, **Function:** getLeftBrackets100000, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out != 0 && SEPFORALL(0, 100000, i, __out + i ↦ '[')

34. **Filename:** baltzo_datafileloader.cpp, **Function:** validateTimeZoneId, **Logic: First Order Separation Logic**, [code]

requires: timeZoneId != 0 && (timeZoneId[0] == '/' ==> res_tmp == -1) && (SEPEXISTS(0, strlen(timeZoneId), i, (timeZoneId + i ↦ sep_v) && !bsl::strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890/_+-", sep_v)) ==> res_tmp == -2) && (SEPFORALL(0, strlen(timeZoneId), i, (timeZoneId + i ↦ sep_v) && bsl::strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890/_+-", sep_v)) ==> res_tmp == 0)
ensures: (__out == -1 ==> timeZoneId[0] == '/') && (__out == -2 ==> SEPEXISTS(0, strlen(timeZoneId), i, (timeZoneId + i ↦ sep_v) && !bsl::strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890/_+-", sep_v))) && (__out == 0 ==> SEPFORALL(0, strlen(timeZoneId), i, (timeZoneId + i ↦ sep_v) && bsl::strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890/_+-", sep_v)))

35. **Filename:** baltzo_localdatetime.cpp, **Function:** baltzo::operator<<, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, object.datetimeTz().size(), j, stream + j ↦ object.datetimeTz().data()[j]) ⋆ SEPFORALL(0, object.timeZoneId().size(), i, stream + object.datetimeTz().size() + i ↦ object.timeZoneId().data()[i]))))

36. **Filename:** baltzo_localtimedescriptor.cpp, **Function:** baltzo::operator<<, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out == stream && (SEPFORALL(0, object.utcOffsetInSeconds().size(), i, stream + i ↦ object.utcOffsetInSeconds().data()[i]) ⋆ SEPFORALL(0, object.dstInEffectFlag().size(), j, stream + object.utcOffsetInSeconds().size() + j ↦ object.dstInEffectFlag().data()[j]) ⋆ SEPFORALL(0, object.description().size(), k, stream + object.utcOffsetInSeconds().size() + object.dstInEffectFlag().size() + k ↦ object.description().c_str()[k]))

37. **Filename:** bdlbb_blobutil.cpp, **Function:** asciiDumpFromBufferStart, **Logic: First Order Separation Logic**, [code]

requires: 0 <= bufferIndex && bufferIndex < source.numDataBuffers() && 0 <= numBytes && numBytes <= source.totalSize() - source.cumulativeSize(buffer

Index)
ensures: __out == stream && (stream.bad() || (stream.good() && SEPFORALL(0, numBytes, i, stream + i ↦ source.buffer(bufferIndex + i / source.buffer(bufferIndex).size()).data()[i % source.buffer(bufferIndex).size()])))

38. **Filename:** bdlde_charconvertutf32.cpp, **Function:** decodeThreeOctets, **Logic: First Order Separation Logic**, [code]

requires: SEPFORALL(0, 3, i, octBuf + i ↦ _)
ensures: __out == ((octBuf[2] & ~k_CONTINUE_MASK) | ((octBuf[1] & ~k_CONTINUE_MASK) << k_CONTINUE_CONT_WID) | ((octBuf[0] & ~k_THREE_OCTET_MASK) << (2 * k_CONTINUE_CONT_WID)))

39. **Filename:** bdlde_charconvertutf32.cpp, **Function:** decodeFourOctets, **Logic: First Order Separation Logic**, [code]

requires: SEPFORALL(0, 4, i, octBuf + i ↦ _)
ensures: __out == ((octBuf[3] & ~k_CONTINUE_MASK) | ((octBuf[2] & ~k_CONTINUE_MASK) << k_CONTINUE_CONT_WID) | ((octBuf[1] & ~k_CONTINUE_MASK) << 2 * k_CONTINUE_CONT_WID) | ((octBuf[0] & ~k_FOUR_OCTET_MASK) << 3 * k_CONTINUE_CONT_WID))

40. **Filename:** bdlde_charconvertutf32.cpp, **Function:** lookaheadContinuations, **Logic: First Order Separation Logic**, [code]

requires: SEPFORALL(0, n, i, (octBuf + i ↦ _))
ensures: SEPFORALL(0, __out, i, (octBuf + i ↦ sep_v && isContinuation(sep_v))) && (__out == n || !isContinuation(*(octBuf + __out)))

41. **Filename:** bdlde_crc32c.cpp, **Function:** calculateCrc32c, **Logic: First Order Separation Logic**, [code]

requires: (length == 0) || (data != 0 && SEPFORALL(0, length, i, data + i ↦ _))
ensures: __out == crc

42. **Filename:** bdlde_sha2.cpp, **Function:** bdlde::operator==, **Logic: First Order Separation Logic**, [code]

requires: lhs.d_bufferSize >= 0 && rhs.d_bufferSize >= 0 && SEPFORALL(0, lhs.d_bufferSize, i, lhs.d_buffer + i ↦ _) && SEPFORALL(0, rhs.d_bufferSize, i, rhs.d_buffer + i ↦ _) && SEPFORALL(0, 8, i, lhs.d_state + i ↦ _) && SEPFORALL(0, 8, i, rhs.d_state + i ↦ _)
ensures: (__out == true ==> (lhs.d_totalSize == rhs.d_totalSize && lhs.d_bufferSize == rhs.d_bufferSize && SEPFORALL(0, lhs.d_bufferSize, i, lhs.d_buffer[i] ↦ rhs.d_buffer[i]) && SEPFORALL(0, 8, i, lhs.d_state[i] ↦ rhs.d_state[i]))) && (__out == false ==> !(lhs.d_totalSize == rhs.d_totalSize && lhs.d_bufferSize == rhs.d_bufferSize && SEPFORALL(0, lhs.d_bufferSize, i, lhs.d_buffer[i] ↦ rhs.d_buffer[i]) && SEPFORALL(0, 8, i, lhs.d_state[i] ↦ rhs.d_state[i])))

43. **Filename:** bdlde_utf8util.cpp, **Function:** get4ByteValue, **Logic: First Order Separation Logic**, [code]

requires: SEPFORALL(0, 4, i, pc + i ↦ _)
ensures: __out == ((*pc & 0x7) << 18) | ((pc[1] & k_CONT_VALUE_MASK) << 12) | ((pc[2] & k_CONT_VALUE_MASK) << 6) | (pc[3] & k_CONT_VALUE_MASK)

44. **Filename:** bdlb_bitstringutil.cpp, **Function:** lt64Raw, **Logic: Propositional Logic**, [code]

requires: 0 <= numBits && numBits < 64
ensures: __out == ((1ULL << numBits) - 1)

45. **Filename:** bdlb_bitstringutil.cpp, **Function:** ge64Raw, **Logic: Propositional Logic**, [code]

requires: 0 <= numBits && numBits < 64
ensures: __out == (~0ULL << numBits)

46. **Filename:** bdlde_utf8util.cpp, **Function:** get2ByteValue, **Logic: Propositional Separation Logic**, [code]

requires: (pc ↦ _) ⋆ (pc + 1 ↦ _)
ensures: __out == ((*pc & 0x1f) << 6) | (pc[1] & k_CONT_VALUE_MASK)

47. **Filename:** bdlde_utf8util.cpp, **Function:** get3ByteValue, **Logic: Propositional Separation Logic**, [code]

requires: pc != nullptr && (pc ↦ _ ⋆ (pc + 1) ↦ _ ⋆ (pc + 2) ↦ _)
ensures: __out == ((*pc & 0xf) << 12) | ((pc[1] & k_CONT_VALUE_MASK) << 6) | (pc[2] & k_CONT_VALUE_MASK)

48. **Filename:** ball_attributecontainerlist.cpp, **Function:** AttributeContainerList::hasValue, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out == true ==> d_head_p != 0) && (__out == false ==> d_head_p == 0 || !d_head_p->d_value_p->hasValue(value))

49. **Filename:** baltzo_zoneinfobinaryreader.cpp, **Function:** decode32, **Logic: Propositional Logic**, [code]

requires: address != nullptr && strlen(address) >= 4
ensures: __out == BSLS_BYTEORDER_BE_U32_TO_HOST(*reinterpret_cast< const int*>(address))

50. **Filename:** bdlbb_blob.cpp, **Function:** bdlbb::operator==, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out == true ==> (lhs.d_buffers == rhs.d_buffers && lhs.d_totalSize == rhs.d_totalSize && lhs.d_dataLength == rhs.d_dataLength && lhs.d_dataIndex == rhs.d_dataIndex && lhs.d_preDataIndexLength == rhs.d_preDataIndexLength)) && (__out == false ==> !(lhs.d_buffers == rhs.d_buffers && lhs.d_totalSize ==

rhs.d_totalSize && lhs.d_dataLength == rhs.d_dataLength && lhs.d_dataIndex == rhs.d_dataIndex && lhs.d_preDataIndexLength == rhs.d_preDataIndexLength))

51. **Filename:** bmqa_queueid.cpp, **Function:** QueueId::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good() && uri().size() >= 0 && correlationId().size() >= 0
ensures: (stream.bad() ==> __out == stream) && (stream.good() ==> (__out == stream && (SEPFORALL(0, uri().size(), j, stream + j ↦ uri().data()[j]) ⋆ SEPFORALL(0, correlationId().size(), i, stream + uri().size() + 1 + i ↦ correlationId().data()[i]))))

52. **Filename:** bmqp_protocolutil.cpp, **Function:** ProtocolUtil::verify, **Logic: Propositional Logic**, [code]

requires: true
ensures: (ci.consumerPriority() == bmqp::Protocol::k_CONSUMER_PRIORITY_INVALID ==> __out == (ci.consumerPriorityCount() == 0)) && (ci.consumerPriority() != bmqp::Protocol::k_CONSUMER_PRIORITY_INVALID ==> __out == (ci.consumerPriorityCount() > 0))

53. **Filename:** bmqst_statcontext.cpp, **Function:** StatContext::valueIndex, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: (__out != -1 ==> SEPEXISTS(0, d_valueDefs_p->size(), i, (*d_valueDefs_p)[i].d_name == name)) && (__out == -1 ==> FORALL(0, d_valueDefs_p->size(), i, (*d_valueDefs_p)[i].d_name != name))

54. **Filename:** bmqp_event.cpp, **Function:** Event::print, **Logic: First Order Separation Logic**, [code]

requires: !stream.bad()
ensures: (stream.bad() ==> __out == stream) && (stream.good() ==> (__out == stream && (SEPFORALL(0, strlen("type"), i, stream + i ↦ "type"[i]) ⋆ (stream + strlen("type") ↦ ':') ⋆ (stream + strlen("type") + 1 ↦ ' ') ⋆ SEPFORALL(0, strlen(type()), j, stream + strlen("type") + 2 + j ↦ type()[j]))))

55. **Filename:** bmqp_eventutil.cpp, **Function:** Flattener::cloneAndPackEachSubQId, **Logic: First Order Separation Logic**, [code]

requires: SEPFORALL(0, subQInfos.size(), i, (subQInfos[i] ↦ sep_v) && (sep_v != 0))
ensures: (__out == rc_SUCCESS) || (__out != rc_SUCCESS ==> (SEPFORALL(0, subQInfos.size(), i, (subQInfos[i] ↦ sep_v) && (sep_v != 0))))

56. **Filename:** bmqp_protocol.cpp, **Function:** PutHeaderFlagUtil::isValid, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out == false ==> (isSet(flags, PutHeaderFlags::e_UNUSED3) || is-

Set(flags, PutHeaderFlags::e_UNUSED4))) && (__out == true ==> !(isSet(flags, PutHeaderFlags::e_UNUSED3) || isSet(flags, PutHeaderFlags::e_UNUSED4)))

57. **Filename:** bmqp_protocol.cpp, **Function:** PushHeaderFlagUtil::isValid, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out == false ==> isSet(flags, PushHeaderFlags::e_UNUSED4)) && (__out == true ==> !isSet(flags, PushHeaderFlags::e_UNUSED4))

58. **Filename:** bmqp_protocol.cpp, **Function:** StorageHeaderFlagUtil::fromString, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: (__out == 0 ==> (out ↦ sep_v && SEPFORALL(0, bdlb::Tokenizer(str, ",").size(), i, (bdlb::Tokenizer(str, ",").begin() + i) != bdlb::Tokenizer(str, ",").end() && StorageHeaderFlags::fromAscii(&sep_v, *(bdlb::Tokenizer(str, ",").begin() + i)) == true && sep_v == (sep_v | *out)))) && (__out == -1 ==> EXISTS(0, bdlb::Tokenizer(str, ",").size(), i, (bdlb::Tokenizer(str, ",").begin() + i) != bdlb::Tokenizer(str, ",").end() && StorageHeaderFlags::fromAscii(&sep_v, *(bdlb::Tokenizer(str, ",").begin() + i)) == false && errorDescription.str().find(*(bdlb::Tokenizer(str, ",").begin() + i)) != std::string::npos))

59. **Filename:** bmqp_queueid.cpp, **Function:** QueueId::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good() && d_id[strlen(d_id)] == '\0' && d_subId[strlen(d_subId)] == '\0'
ensures: __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, strlen(d_id), i, stream + i ↦ d_id[i]) ⋆ SEPFORALL(0, strlen(d_subId), j, stream + strlen(d_id) + j ↦ d_subId[j]))))

60. **Filename:** bmqst_printutil.cpp, **Function:** PrintUtil::printValueWithSeparator, **Logic: First Order Separation Logic**, [code]

requires: stream.good() && groupSize >= 0
ensures: __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, 64, i, (buf + i) ↦ sep_v) ⋆ printValueWithSeparatorImp(buf + 63, value, groupSize, separator))))

61. **Filename:** bmqt_compressionalgorithmtype.cpp, **Function:** CompressionAlgorithmType::isValid, **Logic: First Order Logic**, [code]

requires: str != 0
ensures: (__out == true ==> EXISTS(0, 1, dummy, fromAscii(str) == true)) && (__out == false ==> EXISTS(0, 1, dummy, fromAscii(str) == false))

62. **Filename:** bmqt_correlationid.cpp, **Function:** CorrelationId::print, **Logic: First Order Separation Logic**, [code]

requires: !stream.bad()
ensures: (stream.bad() ==> __out == stream) && (stream.good() ==> (__out

== stream && (SEPFORALL(0, printer.outputSize(), i, stream + i ↦ printer.output()[i])))

63. **Filename:** bmqt_encodingtype.cpp, **Function:** EncodingType::isValid, **Logic: First Order Logic**, [code]

requires: string != 0
ensures: (__out == true ==> EXISTS(0, 1, dummy, fromAscii(string) == true)) && (__out == false ==> EXISTS(0, 1, dummy, fromAscii(string) == false))

64. **Filename:** bmqt_queueflags.cpp, **Function:** QueueFlagsUtil::isValid, **Logic: First Order Separation Logic**, [code]

requires: (!isSet(flags, QueueFlags::e_ADMIN) || /* valid for BlazingMQ admin tasks */) && (isSet(flags, QueueFlags::e_READ) || isSet(flags, QueueFlags::e_WRITE))
ensures: (__out == false ==> SEPEXISTS(0, errorDescription.tellp(), i, errorDescription + i ↦ _)) && (__out == true ==> SEPFORALL(0, errorDescription.tellp(), i, errorDescription + i ↦ old_errorDescription[i]))

65. **Filename:** bmqt_queueoptions.cpp, **Function:** QueueOptions::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good()
ensures: __out == stream && (stream.bad() || (stream.good() && (SEPFORALL(0, strlen("maxUnconfirmedMessages"), i, stream + i ↦ "maxUnconfirmedMessages"[i]) ⋆ SEPFORALL(0, strlen("maxUnconfirmedBytes"), i, stream + strlen("maxUnconfirmedMessages") + 1 + i ↦ "maxUnconfirmedBytes"[i]) ⋆ SEPFORALL(0, strlen("consumerPriority"), i, stream + strlen("maxUnconfirmedMessages") + strlen("maxUnconfirmedBytes") + 2 + i ↦ "consumerPriority"[i]) ⋆ SEPFORALL(0, strlen("suspendsOnBadHostHealth"), i, stream + strlen("maxUnconfirmedMessages") + strlen("maxUnconfirmedBytes") + strlen("consumerPriority") + 3 + i ↦ "suspendsOnBadHostHealth"[i]) ⋆ (SEPEXISTS(0, d_subscriptions.size(), j, stream + offset + j ↦ "Subscriptions:"[j])))))

66. **Filename:** bmqt_version.cpp, **Function:** Version::print, **Logic: Propositional Logic**, [code]

requires: true
ensures: __out == stream

67. **Filename:** bmqu_blob.cpp, **Function:** BlobPosition::print, **Logic: First Order Separation Logic**, [code]

requires: stream.good()
ensures: (stream.bad() ==> __out == stream) && (stream.good() ==> (__out == stream && (SEPFORALL(0, d_buffer.size(), i, stream + i ↦ d_buffer.data()[i]) ⋆ SEPFORALL(0, d_byte.size(), j, stream + d_buffer.size() + j ↦ d_byte.data()[j]))))

68. **Filename:** bmqu_blob.cpp, **Function:** BlobUtil::isValidPos, **Logic: Propositional Logic**, [code]

requires: pos.buffer() >= 0 && pos.byte() >= 0
ensures: (pos.buffer() > blob.numDataBuffers() ==> __out == false) && (pos.buffer() == blob.numDataBuffers() && pos.byte() != 0 ==> __out == false) && (pos.buffer() < blob.numDataBuffers() && pos.byte() >= 0 && pos.byte() < bufferSize(blob, pos.buffer()) ==> __out == true)

69. **Filename:** bmqu_blob.cpp, **Function:** BlobUtil::writeBytes, **Logic: First Order Separation Logic**, [code]

requires: pos.byte() + length <= bufferSize(*blob, pos.buffer())
ensures: (__out != 0) || (__out == 0 && SEPFORALL(0, length, i, (blob->buffer((pos.buffer() + i / bufferSize(*blob, pos.buffer()))).data() + (pos.byte() + i % bufferSize(*blob, pos.buffer()))) ↦ buf[i]))

70. **Filename:** bmqu_printutil.cpp, **Function:** prettyNumberImp, **Logic: First Order Separation Logic**, [code]

requires: buf != nullptr && groupSize > 0 && (separator >= 0 && separator <= 127)
ensures: (__out == buf) && SEPFORALL(0, strlen(buf), i, buf + i ↦ buf[i])

71. **Filename:** bmqu_printutil.cpp, **Function:** prettyBytes, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: __out == stream && (SEPFORALL(0, temp.str().size(), i, stream + i ↦ temp.str().data()[i]))

72. **Filename:** bmqu_printutil.cpp, **Function:** prettyTimeInterval, **Logic: First Order Separation Logic**, [code]

requires: stream.good() && precision >= 0
ensures: __out == stream && stream.good() && (SEPFORALL(0, temp.str().size(), i, stream + i ↦ temp.str()[i]))

73. **Filename:** bmqu_stringutil.cpp, **Function:** removeIfPrecededBySame, **Logic: First Order Separation Logic**, [code]

requires: begin <= end && allowlistBegin <= allowlistEnd && SEPFORALL(0, end - begin, i, (begin + i ↦ _)) && SEPFORALL(0, allowlistEnd - allowlistBegin, j, (allowlistBegin + j ↦ _))
ensures: (__out >= begin && __out <= end) && SEPFORALL(0, __out - begin, i, (begin + i ↦ sep_v) && (!SEPEXISTS(allowlistBegin, allowlistEnd, j, allowlistBegin + j ↦ sep_v) || (i == 0 || *(begin + i - 1) != sep_v)))

74. **Filename:** bmqu_stringutil.cpp, **Function:** StringUtil::startsWith, **Logic: First Order Separation Logic**, [code]

requires: (offset <= str.length()) && SEPFORALL(0, str.length(), i, str.data() + i ↦ _) && SEPFORALL(0, prefix.length(), i, prefix.data() + i ↦ _)
ensures: (__out == true ==> SEPFORALL(0, prefix.length(), i, str.data() + offset + i ↦ prefix.data()[i])) && (__out == false ==> ((offset > str.length()) || ((str.le

ngth() - offset) < prefix.length()) || SEPEXISTS(0, prefix.length(), i, str.data() + offset + i ↦ sep_v && sep_v != prefix.data()[i])))

75. **Filename:** bmqu_stringutil.cpp, **Function:** StringUtil::endsWith, **Logic: First Order Separation Logic**, [code]

requires: true
ensures: (__out == true ==> (str.length() >= suffix.length() && SEPFORALL(0, suffix.length(), i, str[str.length() - 1 - i] == suffix[suffix.length() - 1 - i]))) && (__out == false ==> (str.length() < suffix.length() || SEPEXISTS(0, suffix.length(), i, str[str.length() - 1 - i] != suffix[suffix.length() - 1 - i])))

76. **Filename:** bmqu_stringutil.cpp, **Function:** StringUtil::ltrim, **Logic: First Order Separation Logic**, [code]

requires: str != 0 && SEPFORALL(0, str->size(), i, ((str->begin() + i) ↦ _))
ensures: __out == *str && SEPFORALL(0, __out.size(), i, (__out[i] ↦ sep_v && (sep_v != ' ' || i > 0)))

77. **Filename:** bmqp_protocol.cpp, **Function:** StorageHeaderFlagUtil::isValid, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out == false ==> (isSet(flags, StorageHeaderFlags::e_UNUSED2) || isSet(flags, StorageHeaderFlags::e_UNUSED3) || isSet(flags, StorageHeaderFlags::e_UNUSED4))) && (__out == true ==> !(isSet(flags, StorageHeaderFlags::e_UNUSED2) || isSet(flags, StorageHeaderFlags::e_UNUSED3) || isSet(flags, StorageHeaderFlags::e_UNUSED4)))

78. **Filename:** bmqu_stringutil.cpp, **Function:** StringUtil::squeeze, **Logic: First Order Separation Logic**, [code]

requires: str != nullptr && (SEPFORALL(0, str->size(), i, (*str)[i] ↦ _)) && (characters.size() >= 0)
ensures: __out == *str && !(SEPEXISTS(0, __out.size() - 1, i, (__out[i] == __out[i + 1]) && (characters.find(__out[i]) != bslstl::StringRef::npos)))

79. **Filename:** bmqimp_messagecorrelationidcontainer.cpp, **Function:** MessageCorrelationIdContainer::iterateAndInvoke, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out == true ==> true) && (__out == false ==> true)

80. **Filename:** bmqt_messageguid.cpp, **Function:** MessageGUID::isValidHexRepresentation, **Logic: First Order Logic**, [code]

requires: true
ensures: __out == FORALL(0, MessageGUID::e_SIZE_HEX, i, (buffer[i] >= '0' && buffer[i] <= '9') || (buffer[i] >= 'A' && buffer[i] <= 'F'))

81. **Filename:** bmqt_messageguid.cpp, **Function:** MessageGUID::fromHex, **Logic: First Order Logic**, [code]

requires: strlen(buffer) >= 2 * MessageGUID::e_SIZE_BINARY && FORALL(0, 2 * MessageGUID::e_SIZE_BINARY, i, (buffer[i] >= '0' && buffer[i] <= '9') || (buffer[i] >= 'A' && buffer[i] <= 'F') || (buffer[i] >= 'a' && buffer[i] <= 'f'))
ensures: EXISTS(0, MessageGUID::e_SIZE_BINARY, i, __out.d_buffer[i] == ((k_HEX_INT_TABLE[buffer[2 * i] - '0'] << 4) | (k_HEX_INT_TABLE[buffer[2 * i + 1] - '0'])))

82. **Filename:** bmqio_channelutil.cpp, **Function:** ChannelUtil::handleRead, **Logic: Propositional Logic**, [code]

requires: outPacket->length() == 0
ensures: (__out == 0 || __out == -1) && (__out == -1 ==> outPacket->length() == 0)

83. **Filename:** bmqtst_blobtestutil.cpp, **Function:** BlobTestUtil::toString, **Logic: First Order Logic**, [code]

requires: str != NULL && blob.length() >= 0 && blob.totalSize() >= 0
ensures: (__out == *str) && (__out.length() == blob.totalSize()) && (blob.length() < blob.totalSize() ==> EXISTS(__out.length() - (blob.totalSize() - blob.length()), __out.length(), i, __out[i] == 'X'))

84. **Filename:** bmqvt_propertybag.cpp, **Function:** PropertyBag::import, **Logic: Propositional Logic**, [code]

requires: true
ensures: &__out == this

85. **Filename:** bmqa_abstractsession.cpp, **Function:** AbstractSession::openQueueSync, **Logic: Propositional Logic**, [code]

requires: true
ensures: __out.queueId() == bmqa::QueueId() && __out.result() == bmqt::OpenQueueResult::e_NOT_SUPPORTED

86. **Filename:** bmqa_abstractsession.cpp, **Function:** AbstractSession::configureQueueSync, **Logic: Propositional Logic**, [code]

requires: true
ensures: __out.result() == bmqt::ConfigureQueueResult::e_NOT_SUPPORTED

87. **Filename:** bmqa_abstractsession.cpp, **Function:** AbstractSession::closeQueueSync, **Logic: Propositional Logic**, [code]

requires: true
ensures: __out.queueId() == bmqa::QueueId() && __out.result() == bmqt::CloseQueueResult::e_NOT_SUPPORTED && __out.message() == "Method is undefined in base protocol"

88. **Filename:** bmqa_event.cpp, **Function:** Event::isSessionEvent, **Logic: Propositional Logic**, [code]

requires: (d_impl_sp == nullptr) || (d_impl_sp != nullptr && d_impl_sp->type() == d_impl_sp->type())
ensures: (__out == true ==> (d_impl_sp != nullptr && d_impl_sp->type() == bmqimp::Event::EventType::e_SESSION)) && (__out == false ==> (d_impl_sp == nullptr || d_impl_sp->type() != bmqimp::Event::EventType::e_SESSION))

89. **Filename:** bmqa_event.cpp, **Function:** Event::isMessageEvent, **Logic: Propositional Logic**, [code]

requires: (d_impl_sp == nullptr) || (d_impl_sp != nullptr && d_impl_sp->type() == bmqimp::Event::EventType::e_MESSAGE || d_impl_sp->type() != bmqimp::Event::EventType::e_MESSAGE)
ensures: (__out == true ==> (d_impl_sp != nullptr && d_impl_sp->type() == bmqimp::Event::EventType::e_MESSAGE)) && (__out == false ==> (d_impl_sp == nullptr || d_impl_sp->type() != bmqimp::Event::EventType::e_MESSAGE))

90. **Filename:** bmqa_message.cpp, **Function:** Message::compressionAlgorithmType, **Logic: Propositional Logic**, [code]

requires: isInitialized()
ensures: (rawEvent.isPushEvent() ==> __out == d_impl.d_event_p->pushMessageIterator()->header().compressionAlgorithmType()) && (rawEvent.isPutEvent() ==> __out == d_impl.d_event_p->putMessageIterator()->header().compressionAlgorithmType()) && (!(rawEvent.isPushEvent() || rawEvent.isPutEvent()) ==> __out == bmqt::CompressionAlgorithmType::e_NONE)

91. **Filename:** bmqa_message.cpp, **Function:** Message::ackStatus, **Logic: Propositional Logic**, [code]

requires: isInitialized() && d_impl.d_event_p->rawEvent().isAckEvent()
ensures: __out == bmqp::ProtocolUtil::ackResultFromCode(d_impl.d_event_p->ackMessageIterator()->message().status())

92. **Filename:** bmqa_message.cpp, **Function:** Message::getData, **Logic: Propositional Logic**, [code]

requires: blob != nullptr
ensures: (rawEvent.isPushEvent() ==> __out == d_impl.d_event_p->pushMessageIterator()->loadMessagePayload(blob)) && (rawEvent.isPutEvent() ==> __out == d_impl.d_event_p->putMessageIterator()->loadMessagePayload(blob)) && (!(rawEvent.isPushEvent() || rawEvent.isPutEvent()) ==> __out == -1)

93. **Filename:** bmqeval_simpleevaluator.cpp, **Function:** SimpleEvaluator::Property::evaluate, **Logic: Propositional Logic**, [code]

requires: true
ensures: (__out.isError() ==> context.d_stop == true)

94. **Filename:** bmqimp_event.cpp, **Function:** Event::configureAsMessageEvent, **Logic: Propositional Logic**, [code]

requires: rawEvent.isPushEvent() || rawEvent.isAckEvent() || rawEvent.isPutEvent()
ensures: (__out.type() == EventType::e_MESSAGE) && (__out.d_msgEventMode == MessageEventMode::e_READ)

95. **Filename:** bmqio_resolvingchannelfactory.cpp, **Function:** ResolvingChannelFactory_Channel::peerUri, **Logic: Propositional Separation Logic**, [code]

    requires: d_peerUri != nullptr && d_peerUri ↦ _
    ensures: __out == *d_peerUri

96. **Filename:** bmqma_countingallocatorutil.cpp, **Function:** CountingAllocatorUtil::globalStatContext, **Logic: Propositional Logic**, [code]

    requires: g_initialized
    ensures: __out != 0 && (__out == &g_statContext.object())

97. **Filename:** bmqp_ackmessageiterator.cpp, **Function:** AckMessageIterator::reset, **Logic: Propositional Logic**, [code]

    requires: blob != 0 && blob->length() >= eventHeader.headerWords() * Protocol::k_WORD_SIZE + AckHeader::k_MIN_HEADER_SIZE
    ensures: (__out == rc_SUCCESS) || (__out == rc_INVALID_EVENTHEADER) || (__out == rc_INVALID_ACKHEADER) || (__out == rc_NOT_ENOUGH_BYTES)

98. **Filename:** bmqp_compression.cpp, **Function:** ZLib::writeOutput, **Logic: Propositional Logic**, [code]

    requires: output != NULL && factory != NULL && stream != NULL && errorStream != NULL && &input != NULL
    ensures: (__out == rc_SUCCESS) || (__out == rc_STREAM_INIT_FAILURE) || (__out == rc_STREAM_PROCESS_FAILURE) || (__out == rc_STREAM_END_FAILURE)

99. **Filename:** bmqst_statcontext.cpp, **Function:** convertFromEpoch, **Logic: Propositional Logic**, [code]

    requires: true
    ensures: __out == epochTime - epochOffset()

100. **Filename:** bmqp_protocolutil.cpp, **Function:** ProtocolUtil::ackResultToCode, **Logic: Propositional Logic**, [code]

     requires: true
     ensures: __out == 0 || __out == 1 || __out == 2 || __out == 5 || __out == 6 || __out == 7