Agentic Separation Logic Specification Synthesis

Tarun Suresh [1] **David Korczynski** [1] **Julien Vanegue** [1]

Abstract

Specification synthesis, the task of automatically inferring formal specifications from program implementations and natural language, is important for refactoring, transpilation, optimization, and verification, yet remains an open challenge for large C++ 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, an agentic system for synthesizing expressive, well-validated specifications across large C++ 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 Spec-Agent on open source C++ codebases comprising millions of lines of code. Spec-Agent synthesizes valid specifications for **85%** of target functions, with no false positives observed under fuzzing and expert validation, outperforming Claude Code Opus 4.6 at **10**× 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). High-profile incidents of compromised systems underscore the critical need for verifying the correctness of human-written code, let alone 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 et al., 1936), with many interesting program properties provably unprovable. While LLMs approximate such undecidable tasks with impressive success on benchmarks, their accuracy drops sharply on real-world repositories (Sultan et al., 2026).

In this paper, we address the problem of *code contract* synthesis at scale. 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), (2) security tools can use contracts to guide bug finding, and (3) 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 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-

[1]Bloomberg, New York, NY, USA. Correspondence to: Tarun Suresh <tarsur909@gmail.com, tsuresh10@bloomberg.net>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

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 C++, where no well-established full-program verifier exists and prior verification attempts have proven 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 have applied our system to large, widely-used open source C++ libraries.

In summary, our work makes three contributions:

- Spec-Agent, an end-to-end agentic system that combines counterexample-guided contract inference with automatically generated fuzz harnesses for systematic validation of synthesized specifications.
- 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 on widely-used projects totaling over four million lines of C++ code, in which Spec-Agent synthesizes valid contracts for up to **85%** of target functions with **no** false positives observed under fuzzing and expert validation, outperforming Claude Code Opus 4.6 at **10**× 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 7 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.

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

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 (Floyd, 1993) and Hoare (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.

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., $\texttt{if}\ p\ \texttt{then}\ q_1\ \texttt{else}\ q_2$ 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:

The specification for the `lookup` function on the right side of Figure 2 becomes $post\colon (\forall x \in lst.\ \neg P(x) \Rightarrow ret = \texttt{false}) \vee (\exists x \in lst.\ P(x) \Rightarrow ret = \texttt{true})$, 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

```
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 C.

Logic	Grammar
Prop. (Prop)	p:= true | false | $p \wedge p$ | $p \vee p$ | $\neg p$
First Order (FOL)	fp:= p | $\exists x.fp(x)$ | $\forall x.fp(x)$
Prop. Sep. (Prop SL)	sp:= p | $sp \star sp$ | $x \mapsto n$
FO Sep. (FOSL)	fsp:= sp | $\exists x.fsp(x)$ | $\forall x.fsp(x)$

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

$p \star q$ 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.

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 describe the architecture and workflow of Spec-Agent for inferring expressive specifications at the scale of millions of lines of low-level code (Figure 4).

Code Mining. The first step extracts per-function artifacts from the target repositories. For each function f, Spec-Agent recovers its source code, in-tree documentation comments, and existing unit tests, then computes a set of program features used by all subsequent stages. Static features, presence of loops, conditionals, and induction-variable usage, are computed via Tree-sitter (Tree-sitter) parsing of f. A dynamic feature, whether f accesses the heap, is determined by running the existing tests of f under heap tracing and analyzing the resulting heap profile.

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 a full C++ program verifier.

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). From

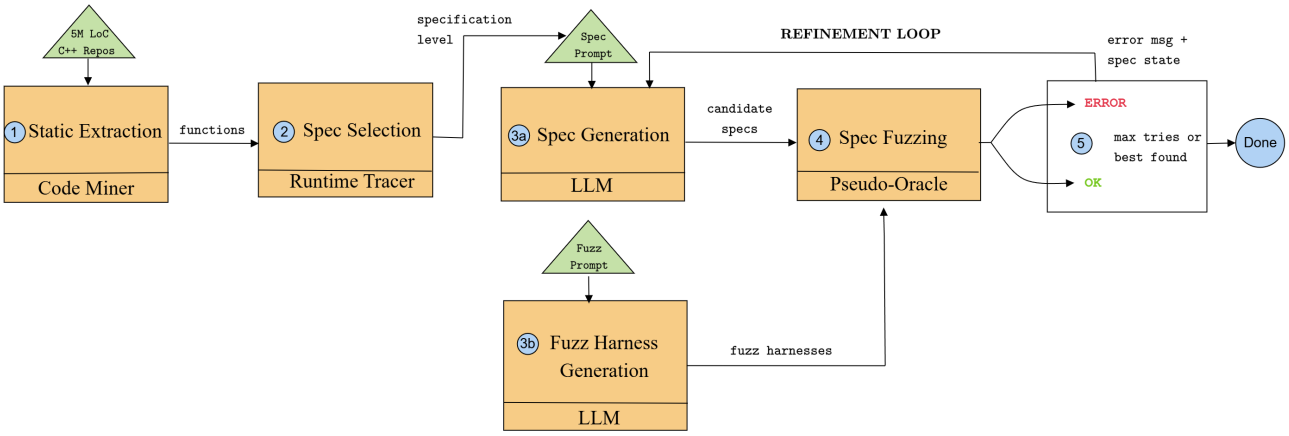


Figure 4. Inside Spec-Agent: Automated fuzz-guided refinement of candidate specifications. The refinement loop iterates until the best possible specification is inferred or the maximum number of retries is reached.

the code-mining features, 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 under libFuzzer. Boolean, arithmetic, and implication operators map to their C++ 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 container/iterator interfaces (e.g., `0` and `v.size()` for a container, `begin` and `end` for an iterator pair). 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. Acceptance requires both that the fuzzer find no counterexample *and* that the candidate be expressed in the targeted logic L. The four languages form a partial order $\sqsubseteq$ depicted in Figure 5: 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 Spec-Agent accepts it if and only if $L \sqsubseteq \ell(cand)$, that is, the candidate reaches at least the target's expressivity. This flexibility 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 $\ell(cand) \sqsubset L$ (e.g., a candidate that omits $\mapsto$ for a heap-touching function targeted at FOSL), the candidate is rejected with a structural diagnostic fed back to the LLM in the next attempt. The loop terminates when the candidate is both fuzz-validated and at the target lattice level, or when the retry budget is exhausted.

4. Evaluation

Experimental Setup. All experiments run on a single machine with an Intel Xeon Silver 4410Y CPU and four NVIDIA H100 GPUs. Spec-Agent is built on top of (Ugare et al., 2026), using vLLM (Kwon et al., 2023) for model serving, libFuzzer for fuzzing, and Clang (Clang) with Tree-sitter for C++ parsing. We evaluate Spec-Agent with four open-weight base models spanning general-purpose and

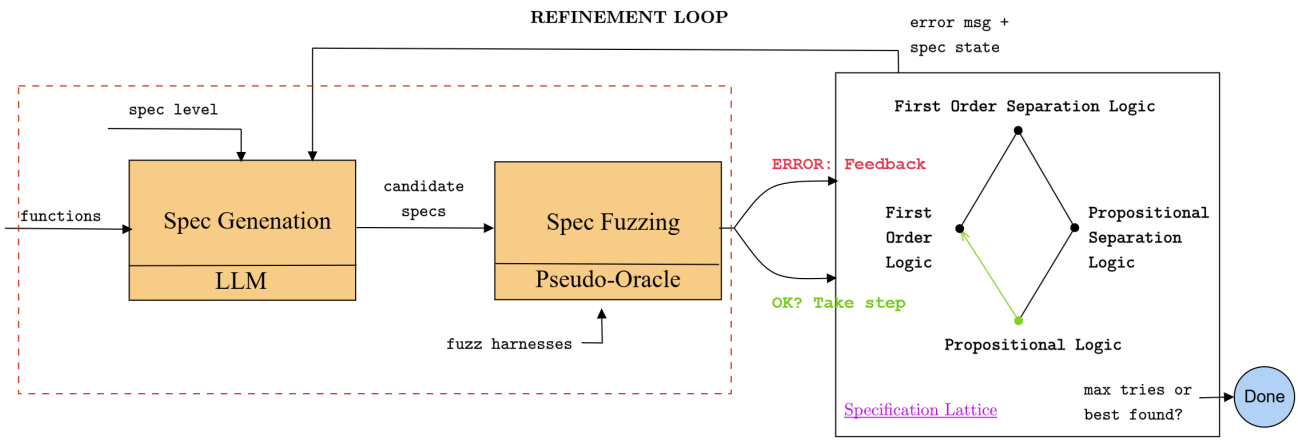


Figure 5. The specification lattice: Spec-Agent maintains an order-theoretic lattice of specification state to track progress in synthesizing the most expressive specification, not only a correct one. Boxes in the dashed red rectangle summarize the full architecture view from Figure 4.

code-specialized families at varying scales: Qwen3.5-35B-A3B (Qwen Team, 2026), Qwen2.5-32B-Instruct (Team, 2024), CWM (team et al., 2025), and Qwen3-Coder-Next (Qwen Team), each decoding greedily with a maximum of 400 tokens per refinement attempt. As a baseline, we run Claude Code (Anthropic, 2024) with Claude Sonnet 4.6 and Claude Opus 4.6 on the same function prompts, unit tests, and fuzz harnesses with identical time and parameter budgets. Claude Code can spawn subagents that synthesize and refine candidate contracts under a budget of 6M weighted tokens per repository, larger than the budget consumed by any model with Spec-Agent. We use Claude Code as the baseline because, as Table 4 shows, no prior specification-synthesis system simultaneously targets large C++ codebases, both pre- and postconditions, and separation logic. To make dollar costs comparable, we report Anthropic's published pricing for Claude and Together AI's pricing (Together AI, 2026) for the open-weight models as a commercial reference point.

Following (Ugare et al., 2026), we evaluate on two large open source C++ repositories: BDE (BDE), a modular C++ library suite of foundational data-structure and utility components used by thousands of developers, and `BlazingMQ` (BlazingMQ) (BMQ), a high-performance, fault-tolerant message-queue library used by thousands of low-latency applications. Both ship with well-documented interfaces and strong test suites. We automatically extract the public functions that carry documentation and existing unit tests, yielding 651 functions for BDE and 508 for BMQ. To complement the automated pseudo-oracle, two experts in separation logic and C++ manually reviewed every generated fuzz harness and a random sample of 100 synthesized specifications per repository; both experts were in agreement. This sample was extracted to create a new benchmark `CPPEVAL` given in Appendix D.

Accuracy. Table 1 compares Spec-Agent against Claude Code Sonnet 4.6 and Opus 4.6 on BDE and BMQ. For each configuration, we report: *Test Valid*, the fraction of functions whose synthesized contract passes every instrumented test execution; *Test Invalid*, the fraction for which the fuzzing pseudo-oracle produces a counterexample; *Compilation Error*, the fraction whose assertion-instrumented build fails; *Trivial*, the fraction of valid contracts that reduce to `true`; and *Avg. Atoms*, the mean number of logical atoms per synthesized contract.

On BDE, the best Spec-Agent configuration (Qwen3-Coder-Next) synthesizes valid specifications for 85.87% of functions, compared to 81.48% for the strongest Claude Code baseline (Claude Opus 4.6) and 78.06% for Claude Sonnet 4.6. On BMQ, the best Spec-Agent configuration (also Qwen3-Coder-Next) reaches 77.73%, compared to 66.69% for Claude Opus 4.6 and 62.64% for Claude Sonnet 4.6. Furthermore, on BDE and BMQ, the best Spec-Agent configurations average 3.35 and 4.15 atoms per contract, respectively, compared to Claude Opus 4.6's 2.34 and 2.46. We corroborate the expressivity claim with a language-level breakdown. Table 2 reports the number of functions with valid specifications for Spec-Agent and Claude Code across both repositories at each specification language. Spec-Agent synthesizes substantially more valid contracts using first-order quantifiers and separation-logic operators, while Claude Code is more heavily skewed toward propositional logic.

Figure 6 characterizes how the number of valid synthesized specifications scales with the number of retry attempts on BDE for Spec-Agent with Qwen3.5-35B-A3B. The curves grow rapidly through the first five attempts, continue to climb with diminishing returns, and effectively plateau by 20 attempts.

We attribute these improvements to Spec-Agent's more

Table 1. Model Performance with Spec-Agent on large C++ repositories

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 correctly synthesized 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

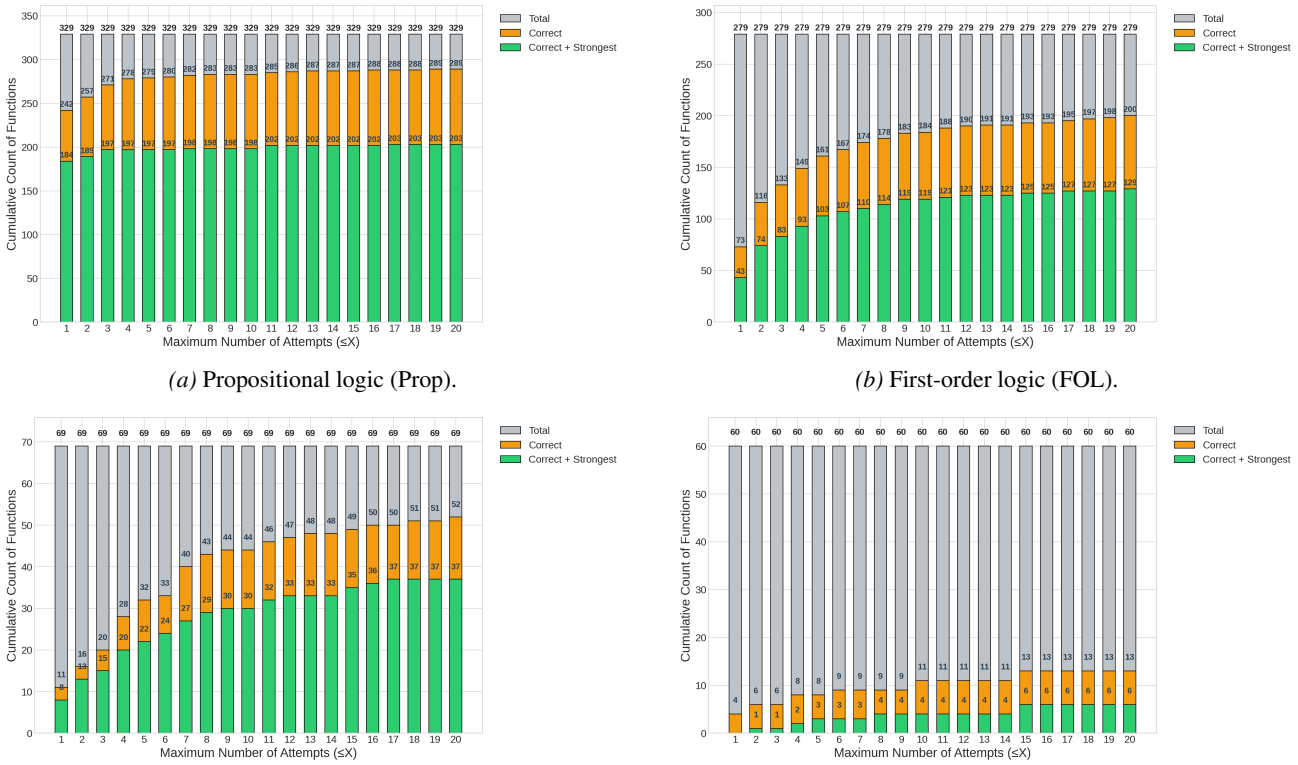
structured refinement loop compared to Claude Code's. Claude Code delegates to subagents that can explore, manage their own context, and write and execute code. This exploration consumes substantially more tokens per step than a focused refinement pass, which, under a fixed compute budget, reduces the number of effective refinement attempts. In contrast, Spec-Agent's loop is deterministic: static and dynamic analyses select the target specification language before synthesis; each attempt makes a single LLM call with the candidate, the prompt, and any prior counterexample; the fuzzer runs after every attempt; and acceptance requires both that the fuzzer report no counterexample and that the candidate be expressed in the targeted logic.

Qualitative Analysis. Figure 6 distinguishes between **correct** contracts (yellow bars) and **strongest** contracts (green bars). Both categories of inferred contracts are correct in that they describe real program behaviors. The strongest contracts describe all possible behaviors of the function, constituting the majority of contracts inferred by Spec-Agent. Spec-Agent achieves this through its specification selection step (Section 3), which ensures that conditional code is represented as disjunctive implications, loops are expressed using quantifiers, and memory operations are captured with separation logic operators. A contract is strongest when it characterizes all expected executions as well as all error paths without exception. While Spec-Agent's performance is strong across most logics, it struggles to discover the strongest contract for the most complex functions where FOSL is selected as the target, as shown in Figure 6d. This may indicate a ceiling in current techniques for these most expressive specification languages, suggesting that new algorithmic insights may be required.

Table 3. 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. A central design choice in Spec-Agent is to generalize the repositories' existing unit tests into fuzz harnesses for more systematic validation of synthesized specifications. Every candidate entering the fuzzing stage has *already* passed the project's existing unit tests under our assertion instrumentation. Despite this prior filter, Table 3 shows the fuzzer still rejects 0.4% of BDE survivors and 8.0% of BMQ survivors, demonstrating the value of fuzzing for improving specification validity beyond fixed test suites, especially for projects with weaker unit-test coverage like BMQ. Each rejected contract corresponds to a counterexample that the project's unit tests alone would have missed, preventing an unsound contract



(a) Propositional logic (Prop).

(b) First-order logic (FOL).

(c) Propositional separation logic (Prop SL).

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

Figure 6. Number of BDE functions with correctly synthesized specifications by number of refinement attempts, for each target specification language. Results shown for Spec-Agent with Qwen3.5-35B-A3B.

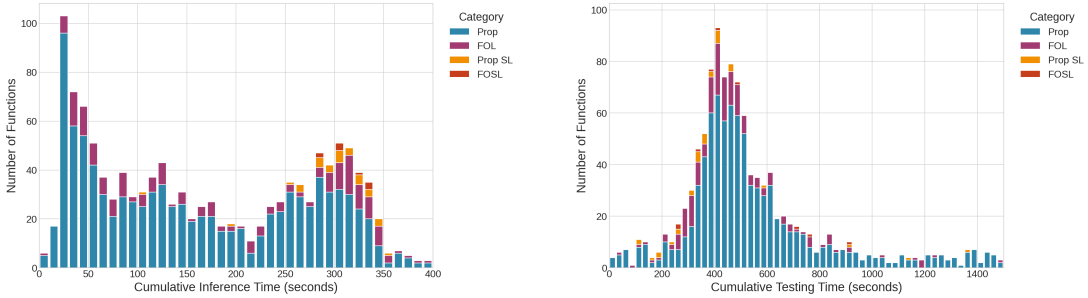
from being accepted.

Figure 7 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. Additional fuzzing-induced coverage measurements for BDE and BMQ are given in Appendix A.

5. Related Work

Earlier work in automated specification synthesis includes Daikon (Ernst et al., 2007), which infers likely pre- and postconditions from execution traces against a fixed invariant grammar, and SLING (Le et al., 2019), which extends this trace-based approach to separation logic for heap-manipulating C programs.

Recent efforts have shifted toward LLM-based specification synthesis, which can leverage natural-language documentation, produce specifications outside any fixed grammar, and incorporate validator feedback into iterative refinement. AutoSpec (Wen et al., 2024) synthesizes pre- and postconditions for C and validates them with the Frama-C theorem prover; SpecGen (Ma et al., 2025) synthesizes specifications for Java using OpenJML and a mutation-and-select stage; ClassInvGen (Sun et al., 2025) synthesizes C++ class invariants using LLM-generated test sequences as the pseudo-oracle; LLM-SE (Liu et al., 2024) synthesizes separation-logic loop invariants from a fine-tuned LLM with symbolic-execution validation; and NL2Postcond (Endres et al., 2024) translates natural-language docstrings into postconditions for Python and Java. Much of this prior work focuses on small benchmark programs rather than real-world codebases. Recently, Fun2Spec (Ugare et al., 2026) scales LLM postcondition synthesis to C++ codebases with millions of lines of code, but synthesizes only postconditions, uses propositional and first-order logic without separation logic, and validates against fixed unit tests only. As Table 4 summarizes, Spec-Agent is the first system that simultaneously scales to large C++ repositories and targets pre- and postcondition synthesis, separation logic, adaptive specification-language selection, fuzzing as a non-fixed pseudo-oracle, and counterexample-guided refinement in a unified pipeline.



(a) Cumulative inference time.

(b) Cumulative testing time.

Figure 7. Distribution of cumulative inference and testing time for each BDE function with an accepted synthesized specification, for Spec-Agent with Qwen3.5-35B-A3B.

Table 4. 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 (Ernst et al., 2007)	✗	✓	✗	✗	✗	✗
SLING (Le et al., 2019)	✗	✓	✓	✗	✗	✗
NL2Postcond (Endres et al., 2024)	✗	✗	✗	✗	✗	✗
LLM-SE (Liu et al., 2024)	✗	✗	✓	✗	✓	✗
AutoSpec (Wen et al., 2024)	✗	✓	✗	✗	✓	✗
ClassInvGen (Sun et al., 2025)	✗	✓	✗	✗	✓	✓
SpecGen (Ma et al., 2025)	✗	✓	✗	✗	✓	✗
Fun2spec (Ugare et al., 2026)	✓	✗	✗	✗	✗	✓
Spec-Agent (our work)	✓	✓	✓	✓	✓	✓

6. 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 employ full C++ verification using a logic prover. The most mature code contract tools do not support C++, and fuzz testing constitutes a strong, scalable alternative that supersedes functional testing, the typical approach used to validate inferred specifications in prior work. Implementing a solver for full C++ program verification is out of scope for this paper.

Strongest Specifications. Our method produced no incorrect specifications under our fuzzing and expert validation regime, with fuzz testing shown to be a cost-effective proxy to full verification in practice. While we instruct the LLM to select logical operators based on the program's syntactic structure, LLMs are not always capable of generating the strongest possible specification capturing the most general behavior of the program (without contingent implementation details). As a result, some inferred contracts may represent only a subset of all possible program behaviors. This subset satisfies all functional tests and fuzz tests employed by Spec-Agent for validation and therefore constitutes a useful approximation. To our knowledge, no existing algorithm can decide whether a given specification for a function is the strongest possible one.

7. Conclusion

Spec-Agent is a new tool for agentic code specification synthesis, tested at scale on millions of lines of C++ code. Spec-Agent synthesizes code contracts in a variety of expressive specification languages for memory-intensive code with loops and conditional reasoning, providing higher accuracy than frontier models at a fraction of the cost. Spec-Agent employs fuzz testing to validate candidate contracts, providing broad code coverage for the synthesized specifications. Automated specification synthesis is key to generating formal code documentation, translating legacy code to modern programming languages, and identifying security vulnerabilities. We expect such capability to become essential as agentic software vulnerability discovery pipelines scale, making automated specification synthesis a necessary countermeasure.

References

Anthropic. Claude code: Command line tool for agentic coding, 2024. URL https://docs.claude.com/en/docs/claude-code.

BDE. https://github.com/bloomberg/bde.

BlazingMQ. A modern, high-performance message queue.

Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. 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.

Clang. clang: a C language family frontend for LLVM.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c: A software analysis perspective. In *International conference on software engineering and formal methods*, pp. 233–247. Springer, 2012.

Endres, M., Fakhoury, S., Chakraborty, S., and Lahiri, S. K. 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.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. 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.

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

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

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., and Chen, J. Security weaknesses of copilot-generated code in github projects: An empirical study, 2025. URL https://arxiv.org/abs/2310.02059.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Le, T. C., Zheng, G., and Nguyen, T. SLING: using dynamic analysis to infer program invariants in separation logic. In McKinley, K. S. and Fisher, K. (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.

Liu, C., Wu, X., Feng, Y., Cao, Q., and Yan, J. Towards general loop invariant generation: A benchmark of programs with memory manipulation. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: 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.

Liu, J., Xia, C. S., Wang, Y., and ZHANG, L. 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.

Ma, L., Liu, S., Li, Y., Xie, X., and Bu, L. 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.

Miller, B., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33:32–44, 12 1990. doi: 10.1145/96267.96279.

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

Qwen Team. Qwen3-coder-next technical report. Technical report. URL https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf. Accessed: 2026-02-03.

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

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

Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In Heiser, G. and Hsieh, W. C. (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.

Sultan, O., Armengol-Estape, J., Kesseli, P., Vanegue, J., Shahaf, D., Adi, Y., and O'Hearn, P. Llms versus the halting problem: Revisiting program termination prediction. *arXiv preprint arXiv:2601.18987*, 2026.

Sun, C., Agashe, V., Chakraborty, S., Taneja, J., Barrett, C. W., Dill, D. L., Qiu, X., and Lahiri, S. K. Classinvgen: Class invariant synthesis using large language models. In Giacobbe, M. and Lukina, A. (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.

team, F. C., Copet, J., Carbonneaux, Q., Cohen, G., Gehring, J., Kahn, J., Kossen, J., Kreuk, F., McMilin, E., Meyer, M., Wei, Y., Zhang, D., Zheng, K., Armengol-Estapé, J., Bashiri, P., Beck, M., Chambon, P., Charnalia, A., Cummins, C., Decugis, J., Fisches, Z. V., Fleuret, F., Gloeckle, F., Gu, A., Hassid, M., Haziza, D., Idrissi, B. Y., Keller, C., Kindi, R., Leather, H., Maimon, G., Markosyan, A., Massa, F., Mazaré, P.-E., Mella, V., Murray, N., Muzumdar, K., O'Hearn, P., Pagliardini, M., Pedchenko, D., Remez, T., Seeker, V., Selvi, M., Sultan, O., Wang, S., Wehrstedt, L., Yoran, O., Zhang, L., Cohen, T., Adi, Y., and Synnaeve, G. Cwm: An open-weights llm for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

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

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

Tree-sitter. An incremental parsing system for programming tools.

Turing, A. M. et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58 (345-363):5, 1936.

Ugare, S., Suresh, T., Misailovic, S., and Vanegue, J. 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.

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

Wang, X., Huang, K., Liang, B., Li, H., and Du, X. 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.

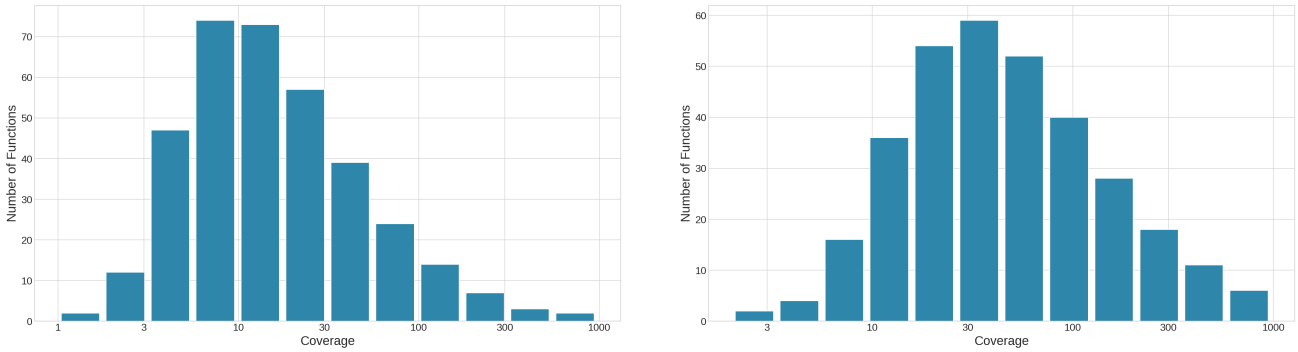
Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S., and Tian, C. Enchanting program specification synthesis by large language models using static analysis and program verification. In Gurfinkel, A. and Ganesh, V. (eds.), *Computer Aided Verification - 36th International Conference, 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.

Yang, K., Zhang, Y., Li, Z., Tao, G., Xu, J., and Liao, X. Harnessagent: Scaling automatic fuzzing harness construction with tool-augmented llm pipelines. *ArXiv*, abs/2512.03420, 2025. URL https://api.semanticscholar.org/CorpusID:283467238.

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL https://www.fuzzingbook.org/.

A. Additional Repository and Fuzzing Statistics

This appendix includes a distributional supplement to the main evaluation showing 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 8. 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 9. 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.

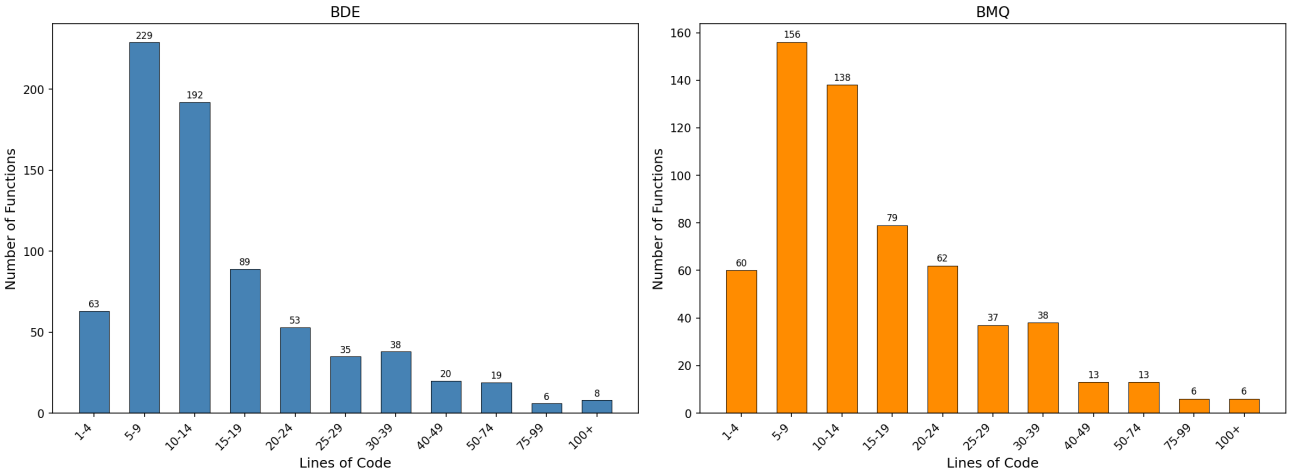


Figure 9. Distribution of lines of code per function for the evaluated functions in BDE (left) and BMQ (right).

B. Cost Breakdown on BDE

Table 5. 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 5 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.

C. 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.

D. Case Study

We list 100 examples of rich C++ specifications synthesized by Spec-Agent on `BDE` (BDE) and `BlazingMQ` (BlazingMQ). 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(bufferIndex)
    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) || isSet(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.length() - 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