Beyond Spec Conformance: A Logic for Validating Stakeholder Expectations

Vasileios Klimis
Queen Mary University of London
London, United Kingdom
v.klimis@qmul.ac.uk

Abstract

Software systems succeed or fail based on their alignment with stakeholder expectations. Yet, while verification rigorously checks conformance to *specifications*, the crucial validation of implicit, nuanced stakeholder expectations remains a pre-formal, ad-hoc activity. This creates a critical validation gap: systems can be formally correct yet functionally misaligned, leading to integration failures and poor user experience. This paper introduces the core idea of Semantic Expectation Logic (SEL), a novel formalism that **elevates stakeholder expectations to become the central, formally-reasoned artifact in the validation process**. SEL provides a logic and framework to systematically elicit, model, and quantify the alignment between what stakeholders expect, what the domain requires, and what the system delivers. Through a motivating case study on REST API conventions, we demonstrate how this expectation-centric approach uncovers validation failures that specification-based methods cannot see. SEL charts a new course for validation, reframing it as a quantifiable engineering discipline of expectation alignment.

ACM Reference Format:
Vasileios Klimis. 2026. Beyond Spec Conformance: A Logic for Validating Stakeholder Expectations. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786821

1 The Validation Stalemate

For decades, software engineering has distinguished between *verification* ("Did we build the system right?") and *validation* ("Did we build the right system?") [6]. While verification has matured into a rigorous discipline with powerful tools for checking spec conformance [7], validation remains a persistent challenge. We are adept at proving a system meets its specification, yet we consistently produce systems that, despite this correctness, are misaligned with their intended purpose and context [15]. This frustrating gap between correctness and "rightness" represents a fundamental stalemate in how we assure software quality.

So, where is the disconnect? We argue that the validation stalemate stems from a critical oversight: the lack of a formal, systematic way to handle the most crucial ingredient of "rightness" – **stakeholder expectations**. The intricate web of assumptions, desired behaviours, and conventions held by developers, users, and operators is the true measure of a system's success [19]. Yet, these expectations are typically relegated to informal documents, ad-hoc user testing, or tribal knowledge – never treated as the first-class, formally-reasoned entities they need to be.

Consider a payment processing API at a financial technology company. The company's API style guide – a document representing a crucial set of shared developer expectations – mandates that all error responses MUST conform to the IETF standard RFC 7807 for machine-readable problem details [18].

The API's formal specification for one endpoint simply states: "If the payment amount exceeds the user's balance, return a '400 Bad Request'." During implementation, a developer handles this case by returning a simple JSON object: `{"message": "Insufficient funds"}` with a '400' status. Automated tests are written to verify this exact behaviour, and they pass. The system is **100% compliant with its endpoint-specific specification.**

However, when another team tries to integrate with this API, their client – built to expect standardised RFC 7807 errors – fails. The client developer's **expectation**, based on the mandatory company-wide style guide, was for a structured body like:

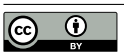
```
{
  "type": "/errors/insufficient-funds",
  "title": "Insufficient Funds",
  "status": 400,
  "detail": "The transaction amount exceeds the
            user's available balance."
}
```

This is a classic validation failure. The root cause is not a bug in the code's logic, but a **semantic misalignment**. The system adheres to its local specification but violates the broader, more critical expectation established by the company's shared "domain rules" (the style guide). Verification passed, but validation failed, leading to costly integration rework and inconsistency across the company's microservices.

2 A New Approach: Expectation as a Formal Artifact

To break this stalemate, we propose a new idea: *to elevate stakeholder expectations from an informal afterthought to the central, formal artifact of the validation process.* We introduce **Semantic Expectation Logic (SEL)**, a framework built on this premise. SEL's novelty lies in its ability to formally model and reason about the alignment between three distinct perspectives, as illustrated in Figure 1:

- **Domain Semantics ($\mathcal{D}$):** The ground truths and rules of the operational context (e.g., HTTP standards).



ICSE-NIER '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2425-1/26/04
https://doi.org/10.1145/3786582.3786821

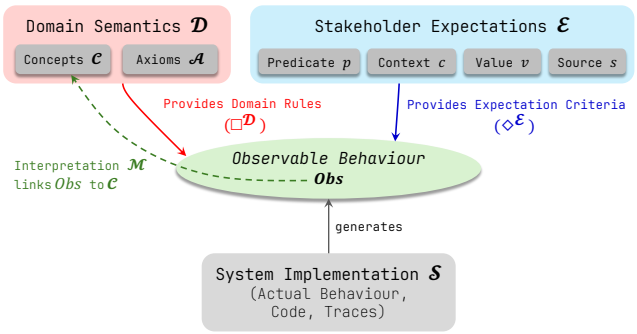


Figure 1: The SEL conceptual model, centered on reasoning about Expectations.

- **Stakeholder Expectations ($\mathcal{E}$):** The (often implicit, contextual, or conflicting) beliefs about how the system *should* behave, captured as a formal, (though possibly inconsistent) set.
- **System Behaviour ($\mathcal{S}$):** The actual, observable behaviour of the system, typically derived from execution traces or logs.

The power of this separation is in untangling different sources of truth. By treating $\mathcal{E}$ as a distinct entity, we can analyse misalignments that are not strictly implementation bugs (where $\mathcal{S}$ violates a spec) nor domain errors (where $\mathcal{S}$ violates $\mathcal{D}$), but are instead crucial failures of alignment between $\mathcal{S}$ and $\mathcal{E}$.

2.1 Logical and Semantic Foundations

Reasoning about the "messy middle" of expectations requires a logic more nuanced than classical binary truth. SEL is therefore built on two key logical foundations that are essential for handling the realities of software validation.

First, to manage ambiguity and incomplete information, SEL employs a *trivalent logic* based on the set {⊤(satisfied), ⊥(violated), ∘(unknown)}. This approach, drawing from foundational work in many-valued logic [10], is critical for a practical validation framework. Unlike binary logic which forces a premature true/false judgment, the ∘ (unknown) value allows the engine to soundly acknowledge what cannot be determined from available evidence, such as from incomplete traces or ambiguously worded expectations. This prevents the system from making unsound claims of satisfaction or violation.

Second, to distinguish the authority and nature of different statements, SEL introduces concepts from modal logic [5, 9]. Modal logic is designed to reason about different modes of truth, such as necessity and possibility. SEL adapts this by defining three distinct modalities to represent the different semantic perspectives:

- $\Box^{\mathcal{D}}\phi$: A necessity modality read as "ϕ is **required** by the domain". This is the strongest form of truth, representing an inviolable axiom.
- $\Diamond^{\mathcal{E}}\phi$: A possibility-like modality read as "ϕ is **expected** by stakeholders". This represents a desired, but not necessarily universal, truth. Its satisfaction is the core of validation.
- $\blacksquare^{\mathcal{S}}\phi$: Another necessity modality read as "ϕ is **guaranteed** by the system". This represents an empirical truth derived from observing the system's behaviour.

This logical machinery provides a precise language for expressing complex validation scenarios. An unmet expectation is no longer an informal complaint, but the formal, checkable statement $\Diamond^{\mathcal{E}}\phi \wedge \neg\blacksquare^{\mathcal{S}}\phi$. A system that is correct but surprising is captured by $\blacksquare^{\mathcal{S}}\phi \wedge \neg\Diamond^{\mathcal{E}}\phi$. This formal grounding is what enables the systematic and quantifiable approach to validation presented in this paper.

3 Operationalising the SEL Framework

The SEL framework is operationalised through two key mechanisms: a systematic process for constructing the Expectation Set ($\mathcal{E}$), and a suite of metrics for quantifying alignment.

3.1 Eliciting Expectations via the EMA Process

The practical utility of SEL hinges on systematically constructing the Expectation Set ($\mathcal{E}$) from latent stakeholder knowledge. To this end, we propose the *Expectation Mining Algorithm (EMA) process*. This human-in-the-loop, iterative approach is inspired by query-based learning from computer science theory [4] and active learning from machine learning [21], but is uniquely tailored for eliciting subjective, often inconsistent, human knowledge.

The process involves presenting concrete system behaviours (observations) to stakeholders and querying their judgment on the expectedness of each behaviour in its specific context. Through this guided dialogue, stakeholders articulate and formalise their tacit knowledge into a structured set, where each expectation is captured not just as a predicate, but is annotated with its application context, a stakeholder-assigned confidence value ($v \in [0, 1]$), and its source.

3.2 Quantifying Alignment with SEL Metrics

With the three perspectives defined, SEL enables quantitative assessment through a suite of novel validation metrics. These provide a dashboard for assessing semantic alignment, moving beyond traditional measures like test coverage [2]. Key metrics include:

- *Weighted Expectation Coverage* (*WEC*): Measures the fraction of total "expected value" (weighted by stakeholder confidence v) that the system actually satisfies.
- *Surprise Factor (SU):* A novel metric designed to directly quantify developer friction and the "pain" of misalignment. Calculated as the complement of *WEC* ($1 - WEC$), it represents the proportion of total "expected value" that the system fails to deliver, providing a more intuitive engineering signal than a simple failure count.
- *Semantic Fidelity* (*SF*): Measures the system's conformance to the fundamental, inviolable axioms of its operational domain ($\mathcal{D}$).
- *Stakeholder Consensus (SC):* Measures the level of agreement among different stakeholders on shared expectations.

The ability to distinguish between a domain violation (low *SF*) and an expectation misalignment (high *SU*) is a core diagnostic feature of our approach.

4 SEL in Action: A Case Study on API Conventions

To provide emerging evidence for our approach, we conducted a case study with a clear objective: *to determine if SEL could identify and quantify validation gaps in a system that was fully compliant with its local specification.* This study serves not as an exhaustive validation of a production system, but as a controlled, illustrative demonstration of SEL's core mechanisms and the insights it can provide. For this purpose, we developed a mock RESTful API designed

to be *100% compliant with a baseline OpenAPI specification* while intentionally violating several common developer conventions.

Applying the SEL framework began by defining a **Domain Model ($\mathcal{D}$)** of core HTTP concepts and generating 21 interaction traces to represent System Behaviour ($\mathcal{S}$). To form the crucial **Expectation Set ($\mathcal{E}$)**, we then applied the EMA process (§3). An author systematically simulated three stakeholder roles (S1: a developer expecting strict conventions; S2: focused on basic success paths; S3: with differing error views). Using our prototype tool, each of the 21 traces was presented, displaying the full HTTP request and response. The simulated stakeholder provided a 'Yes/No/Unclear' judgment and a confidence value. This systematic process yielded 15 formal expectations (derived from 'Yes' responses where a predicate was formulated), while 'No' responses were recorded as explicit violations and 'Unclear' responses were logged as points of ambiguity for future analysis.

The analysis of these artifacts with our prototype engine produced the results summarised in Table 1. The most telling result is the stark contrast between the near-perfect Semantic Fidelity and the high Surprise Factor. This provides clear, empirical evidence that a system can be compliant with its domain's fundamental rules and basic specification, yet simultaneously be deeply misaligned with the conventional expectations of its users. This demonstrates SEL's core value: illuminating the validation gap that traditional, specification-based verification is not equipped to address.

Table 1: SEL Validation Metrics for the Mock API

Metric	Mock API Value
EC	0.40
WEC	0.42
SF	0.89
SU	0.58
SC	0.83

The high Surprise Factor ($SU = 0.58$) is driven by specific, impactful validation gaps. Our engine pinpointed numerous violations, including:

- *Incorrect Error Handling for Duplicates:* Instead of the expected `409 Conflict` or a general `4xx` client error, the API returned a misleading `500 Internal Server Error` for a duplicate POST attempt (`trace_004`). This violated high-confidence expectations from multiple stakeholders.
- *Violation of Domain Axioms:* Beyond expectation failures, the engine flagged a violation of a fundamental domain rule. The `201 Created` response in `trace_002` did not include the mandatory `Location` header, directly violating a core HTTP convention (Axiom A06). This specific failure is the cause of the imperfect Semantic Fidelity score ($SF = 0.89$).
- *Unconventional PUT Semantics:* The API's PUT method performed an "upsert" (creating a resource if it did not exist) and returned a `200 OK`. This single behaviour violated two distinct stakeholder expectations: one that expected the operation to fail with a `404 Not Found`, and another that expected an upsert to return a `201 Created` status.

(The full list of 15 expectations and their evaluation status is available in our supplementary material.)

These findings are not merely style critiques; they represent tangible friction points that complicate client development. By systematically processing elicited expectations against observed behaviour, SEL transforms subjective feelings of "this API is awkward" into specific, evidence-backed validation failures with quantifiable impact indicators.

5 Related Work

SEL builds upon and distinguishes itself from several established research areas.

Formal Verification and Specification Mining. Traditional Formal Verification (FV) techniques like model checking [7, 9] excel at proving conformance between an implementation and a formal specification. FV is concerned with "building the system right". SEL, in contrast, is a validation framework concerned with "building the right system", focusing on alignment with stakeholder expectations, which may not be captured in any specification. In this, SEL is also distinct from Specification Mining [3, 16], which infers behavioural models from what a system *does*. SEL's EMA process elicits what stakeholders *believe it should do*, providing an orthogonal source of truth for validation.

Requirements Engineering and Software Testing. SEL shares its primary goal with Requirements Engineering (RE) – capturing stakeholder intent [19, 25]. However, SEL provides a formal backend for reasoning about elicited expectations and a quantitative feedback loop (via metrics) that is absent in many traditional RE approaches. Our EMA process, inspired by query-based learning [4], offers a systematic elicitation technique within the broader RE landscape. Compared to traditional Software Testing, which often focuses on functional correctness against a spec or code coverage [2], SEL provides a formal basis for generating and evaluating test cases that target semantic and conventional alignment.

Validation of Complex Semantics. While sophisticated techniques exist to validate the deep semantics of compilers [13, 26], GPU representations [11], and memory models [1, 12] against machine-derived specifications, SEL provides the first formal framework for validating a system against a different source of semantic truth: the elicited and quantified expectations of its human stakeholders.

Related Formalisms and Methodologies. SEL's formalism is grounded in established logical principles. The use of a trivalent logic draws from foundational work in Multi-Valued Logic [10], and its application to handle ambiguity in SE has precedent. For instance, Modal Transition Systems (MTS) use a three-valued logic to analyse systems with 'may' and 'must' modalities, with tool support like the MTSA platform [8] for merging and reasoning about different viewpoints [22, 23]. Similarly, our use of modal operators builds on Modal Logic [5] to distinguish different sources of truth. While SEL shares this formal heritage, it differs from methodologies like Behavior-Driven Development (BDD) [17] or Goal-Oriented Requirements Engineering (GORE) [24]. Whereas BDD and GORE guide upfront design based on goals, SEL provides a framework for the continuous, quantitative validation of a running system's concrete behaviours against elicited, low-level expectations.

6 Discussion and The Path Forward

The results from our case study, while preliminary, suggest that formalising stakeholder expectations offers a new and powerful lens for software validation. This section discusses the practical implications of this approach for engineering teams, the significant challenges that remain, and the threats to the validity of our initial findings.

6.1 From Formalism to Engineering Practice

While grounded in logic, the core idea of SEL is deeply practical. It provides engineers with a concrete framework to tackle pervasive, yet often intangible, problems in software development:

- *It makes "developer experience" a measurable quantity.* Terms like "awkward API" or "confusing behaviour" are notoriously difficult to act upon. SEL, through metrics like the Surprise Factor (SU), translates these subjective feelings into a quantifiable signal that can be tracked, prioritised, and used to justify engineering effort on non-functional improvements.
- *It provides a shield against implicit assumptions.* The EMA process forces assumptions—from both stakeholders and developers—out into the open. By creating a shared, explicit artifact of expectations ($\mathcal{E}$), teams can reduce the "it works on my machine" class of problems that often stem from divergent mental models of how a component should behave, especially in microservice architectures.
- *It elevates documentation to a testable artifact.* Company style guides, API documentation examples, and tutorials are often treated as passive documents. SEL allows them to be treated as an active part of the validation suite. An expectation can be directly derived from a documentation example, and if the system's behaviour contradicts that example, a validation failure is flagged. This creates a powerful incentive for keeping documentation and implementation in sync.

SEL provides a language and a process to have more precise conversations about what "good" software means, moving beyond just the narrow definition of "bug-free".

6.2 Implications: A Discipline of Alignment

SEL is not merely a new testing technique; it proposes a shift towards a continuous, quantifiable discipline of semantic alignment.

- *Making Conventions First-Class Citizens:* In domains like API design, unstated conventions are as important as formal specifications [20]. SEL provides a mechanism to make these conventions explicit, formal, and, most importantly, *testable*. The high Surprise Factor (SU) in our case study was a direct measure of these conventional violations, an entire class of issues invisible to standard spec-based verification.
- *A Language for Design Trade-offs:* Real-world engineering involves trade-offs. The ability to model conflicting expectations (e.g., E005 vs. E005b on PUT behaviour) and measure the degree of disagreement with metrics like Stakeholder Consensus (SC) provides a formal basis for discussing these trade-offs. A low SC score becomes concrete evidence that a design decision requires explicit documentation or negotiation, moving the conversation from informal argument to data-driven prioritisation.
- *From Ad-Hoc to Systematic Elicitation:* The EMA process, even in its pragmatic form, offers a systematic alternative to traditional, often unstructured, requirements gathering [19]. By grounding the dialogue in concrete system behaviours, it forces a clearer conversation between developers and stakeholders about what is truly expected, reducing the ambiguity that plagues natural language specifications.
- *A New Target for Testing and Analysis:* SEL provides a new, formal target for testing activities. Instead of just writing tests to verify a specification, engineers can now write tests to explicitly validate or falsify high-value stakeholder expectations from the set $\mathcal{E}$. This suggests a new form of "expectation-driven testing", where test suites are evaluated not just on code coverage [2], but on their *Expectation Coverage*, ensuring that what matters to stakeholders is actually being tested.

6.3 Challenges and Future Work

As a new idea, SEL presents numerous challenges and avenues for future research. The path forward involves tackling three key areas to realise its full potential:

(1) *Automation: From Human-in-the-Loop to Human-on-the-Loop:* While effective, the current EMA process requires significant human effort for predicate formulation and behaviour sampling. The path to scalability involves shifting the stakeholder's role from being *in* the loop to being *on* the loop – supervising and correcting an automated process. Future work should explore using Natural Language Processing (NLP) to automatically suggest predicate structures from violation reasons, and employing active learning strategies [21] to intelligently propose the most ambiguous or contentious system behaviours for human review.
(2) *Richer Predicate Logic and Analysis:* The prototype's heuristic interpretation of predicate text is a key limitation. The full power of SEL will be unlocked by developing a more expressive, perhaps domain-specific, language (DSL) for writing expectations. This would enable the formal validation of more complex, stateful properties like full idempotency or temporal safety properties, potentially connecting SEL's frontend with the analytical power of established backend formalisms like temporal logic or model checking [7].
(3) *Tooling Integration and Continuous Validation:* To be effective in practice, SEL must integrate into standard development workflows. We envision plugins for CI/CD pipelines that automatically run SEL checks on test results and feed the metrics into developer dashboards. This would transform validation from a late-stage, manual activity into a continuous, automated assessment of semantic alignment, similar to how Runtime Verification (RV) provides continuous conformance checking [14].

6.4 Threats to Validity

Our findings should be considered in light of several limitations common to preliminary research. *Construct Validity* is threatened by our use of simulated stakeholders and the engine's heuristic interpretation of user-defined predicate text. *Internal Validity* is impacted by author bias in the design of the mock API and its violations. *External Validity* is limited as the study uses a single mock API in the REST domain with a finite set of 21 traces, so findings

may not generalise. Finally, regarding *Conclusion Validity*, our study is primarily demonstrative; the quantitative results illustrate SEL's potential, not a definitive statistical validation.

7 Conclusion

This paper introduced Semantic Expectation Logic (SEL), a new idea that reframes validation as a formal, quantifiable discipline of expectation alignment. By elevating stakeholder expectations to a central, formally-reasoned artifact, SEL provides a framework to systematically identify and measure the critical semantic gaps between specified behaviour and stakeholder intent. Our emerging results show that this expectation-centric approach can uncover validation failures that traditional methods cannot see. SEL charts a promising course for building systems that are not only correct, but are provably aligned with what their stakeholders truly expect.

Data and Artifact Availability. Our prototype implementation, mock API, data files, and reproduction instructions for the case study are publicly available on Zenodo DOI 10.5281/zenodo.15850659. The artifact has been anonymised for review.

References

[1] Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024. Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 341 (2024). doi:10.1145/3689781

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2nd ed.). Cambridge University Press.

[3] Glenn Ammons, Rastislav Bodik, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. 4–16. doi:10.1145/503272.503275

[4] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[5] Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science, Vol. 53. Cambridge University Press.

[6] B.W. Boehm. 1981. *Software Engineering Economics*. Prentice-Hall. https://books.google.co.uk/books?id=VphQAAAAMAAJ

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. The MIT Press.

[8] Nicolas D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastian Uchitel. 2008. MTSA: The Modal Transition System Analyser. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 475–476. doi:10.1109/ASE.2008.78

[9] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems* (2nd ed.). Cambridge University Press.

[10] Stephen Cole Kleene. 2002. *Mathematical Logic*. Dover Publications. Reprint of the 1967 Wiley edition. Discusses many-valued logics..

[11] Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2023. Taking Back Control in an Intermediate Representation for GPU Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 60 (Jan. 2023), 30 pages. doi:10.1145/3571253

[12] Vasileios Klimis, Alastair F. Donaldson, Viktor Vafeiadis, John Wickerson, and Azalea Raad. 2024. Challenges in Empirically Testing Memory Persistency Models. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*. New York, NY, USA. doi:10.1145/3639476.3639765

[13] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. doi:10.1145/2594291.2594334

[14] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. doi:10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[15] Nancy G. Leveson. 1995. *Safeware: System Safety and Computers*. Addison-Wesley Professional.

[16] David Lo and Michael D. Ernst. 2011. Specification Mining: A Survey. *ACM SIGSOFT Software Engineering Notes* 36, 5 (2011), 1–6. doi:10.1145/2020976.2020983

[17] Dan North. 2006. Introducing BDD. https://dannorth.net/introducing-bdd/.

[18] M. Nottingham and E. Wilde. 2016. Problem Details for HTTP APIs. doi:10.17487/RFC7807

[19] Bashar Nuseibeh and Steve Easterbrook. 2000. Requirements Engineering: A Roadmap. *Future of Software Engineering (FOSE '00)* (2000), 35–46. doi:10.1145/336512.336523

[20] Chris Richardson. 2018. *Microservices Patterns: With examples in Java*. Manning Publications.

[21] Burr Settles. 2009. *Active Learning Literature Survey*. Technical Report Computer Sciences Technical Report 1648. University of Wisconsin-Madison. Widely cited survey..

[22] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. 2009. Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Transactions on Software Engineering* 35, 3 (2009). doi:10.1109/TSE.2008.107

[23] Sebastian Uchitel and Marsha Chechik. 2004. Merging partial behavioural models. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct. 2004), 43–52. doi:10.1145/1041685.1029904

[24] A. van Lamsweerde. 2001. Goal-oriented requirements engineering: a guided tour. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*. 249–262. doi:10.1109/ISRE.2001.948567

[25] Axel van Lamsweerde. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532