Self-Improving Agentic Development via OpenCyc Knowledge Representation: A Framework for Machine-Verified Autonomous Software Engineering

Anonymous Author
Institution Affiliation
email@institution.edu

July 24, 2025

Abstract

We propose a novel framework for autonomous software development that combines formal knowledge representation (OpenCyc), large language models (LLMs), and machine verification systems to create self-improving development agents. Unlike current AI coding assistants that operate reactively, our approach enables agents to reason formally about software architecture, development priorities, and code quality using a comprehensive knowledge base of software engineering principles. The agent maintains explicit representations of its own codebase, development state, and architectural decisions, enabling autonomous planning, implementation, and verification cycles. We demonstrate how this approach naturally integrates with existing formal verification tools (ACL2, Lean4, Coq) while providing a practical path toward fully autonomous software development.

1 Introduction

Current AI-assisted software development operates primarily in a reactive mode: human developers request code generation, debugging assistance, or architectural advice from LLMs like GPT-4 or Claude. While powerful, this paradigm suffers from several fundamental limitations:

1. **No persistent memory** of architectural decisions or development history
2. **Lack of formal reasoning** about software correctness and design principles
3. **Human-driven planning** with no autonomous goal-setting or priority management
4. **Isolated interactions** without cumulative learning or self-improvement

Meanwhile, the formal verification community has developed sophisticated tools (ACL2, Lean4, Coq, Isabelle/HOL) for machine-verified software development, but these require extensive human expertise and manual theorem proving.

We propose bridging these domains through **OpenCyc-based agentic development**: autonomous agents that maintain formal knowledge representations of software engineering principles, their own codebase, and development processes. This enables genuinely autonomous software development with machine-verifiable correctness guarantees.

1.1 Key Contributions

- A formal framework for representing software engineering knowledge in OpenCyc
- An autonomous development cycle that reasons about priorities, implements solutions, and verifies correctness
- Integration pathways with existing formal verification systems

- A practical implementation demonstrating self-improving knowledge base development
- Theoretical foundations for provably correct autonomous software engineering

2 Background and Motivation

2.1 Limitations of Current AI Development Tools

Current AI coding assistants exhibit several critical limitations for autonomous development:

Statelessness: Each interaction begins from scratch, with no memory of previous architectural decisions, bug fixes, or design rationale.

Reactive Planning: Tools respond to human requests but cannot autonomously identify technical debt, architectural improvements, or feature priorities.

Informal Reasoning: Code generation relies on pattern matching and statistical inference rather than formal reasoning about correctness, performance, or maintainability.

No Self-Modification: AI tools cannot autonomously improve their own capabilities or development processes.

2.2 The Promise of Formal Knowledge Representation

OpenCyc provides a mature framework for representing and reasoning about complex domains through:

- **Formal Ontologies**: Structured representations of concepts and relationships
- **Logical Inference**: Sound reasoning about complex relationships and dependencies
- **Meta-Reasoning**: The ability to reason about reasoning itself
- **Persistent Knowledge**: Cumulative learning and knowledge retention

Applied to software development, this enables agents to:

- Maintain formal models of software architecture and design patterns
- Reason about feature dependencies, implementation priorities, and technical debt
- Learn from development experience and encode best practices
- Make provably sound decisions about code modifications

2.3 Machine Verification and Autonomous Development

Formal verification systems like ACL2 and Lean4 provide mathematical guarantees about software correctness but require significant human expertise. Our framework creates a pathway for autonomous agents to:

1. **Generate verification conditions** based on formal specifications
2. **Attempt automated proofs** using theorem proving tactics
3. **Learn from proof failures** to improve code generation
4. **Maintain verification invariants** across code modifications

3 The OpenCyc Agentic Development Framework

3.1 Core Architecture

Our framework consists of five interconnected components (Figure 1):

Agent Core (Planning/Exec)	↔	OpenCyc KB (Dev Knowledge)
	↕	
Code Generator (LLM Integration)	Repository Manager (Git)	Test Executor (Automated QA)

Figure 1: Core architecture of the agentic development framework

Agent Core: The central reasoning system that queries the knowledge base, makes development decisions, and coordinates other components.

OpenCyc Knowledge Base: Formal representations of software engineering principles, current codebase structure, development history, bug patterns, and performance characteristics.

Verification Engine: Integration with formal verification tools for generating and checking proofs of correctness.

Code Generator: LLM-based code synthesis guided by formal specifications and architectural constraints.

Repository Manager: Version control integration for tracking changes, managing branches, and maintaining development history.

Test Executor: Automated testing framework providing feedback for the learning cycle.

3.2 Knowledge Representation Schema

We define a comprehensive ontology for software development knowledge:

3.2.1 Architectural Knowledge

```
(#$isa #$SoftwareArchitecture #
    $AbstractionLevel)
(#$isa #$LayeredArchitecture #
    $SoftwareArchitecture)
(#$isa #$MicroserviceArchitecture #
    $SoftwareArchitecture)

(#$implies
  (#$and (#$uses ?system #
      $LayeredArchitecture)
          (#$dependsOn ?upperLayer ?
              lowerLayer))
  (#$not (#$dependsOn ?lowerLayer ?
      upperLayer)))
```

Listing 1: Architectural knowledge representation

3.2.2 Development Process Knowledge

```
(#$isa #$DevelopmentPhase #
    $TemporalThing)
(#$isa #$RequirementAnalysis #
    $DevelopmentPhase)
(#$isa #$Implementation #
    $DevelopmentPhase)
(#$isa #$Testing #$DevelopmentPhase)

(#$implies
  (#$and (#$currentPhase ?project ?phase
      )
          (#$testsFailing ?project))
  (#$shouldTransitionTo ?project #
      $Debugging))
```

Listing 2: Development process representation

3.2.3 Code Quality Knowledge

```
(#$isa #$CodeQualityMetric #
    $MeasurableProperty)
(#$isa #$CyclomaticComplexity #
    $CodeQualityMetric)
(#$isa #$TestCoverage #
    $CodeQualityMetric)

(#$implies
  (#$and (#$cyclomaticComplexity ?
      function ?complexity)
          (#$greaterThan ?complexity 10))
  (#$shouldRefactor ?function))
```

Listing 3: Code quality metrics

3.3 Autonomous Development Cycle

The agent operates in continuous development cycles:

Algorithm 1 Autonomous Development Cycle

1: **Phase 1:** Query current development state
2: **Phase 2:** Reason about development priorities
3: **Phase 3:** Generate implementation strategy
4: **Phase 4:** Generate and verify code
5: **Phase 5:** Execute tests and analyze feedback
6: **Phase 6:** Integrate knowledge and update KB
7: **GOTO** Phase 1

3.3.1 State Assessment

```
;; Query current development state
(#$currentPhase #$MyProject ?phase)
(#$testResults #$MyProject ?results)
(#$technicalDebt #$MyProject ?debt)
(#$featureRequests #$MyProject ?requests
    )
```

Listing 4: State assessment queries

3.3.2 Priority Reasoning

```
;; Determine next development priority
(#$shouldWorkOn #$MyProject ?task)
(#$because
  (#$and (#$criticalBug ?task)
         (#$affectsUsers ?task #
             $ManyUsers))
  (#$priority ?task #$Highest))
```

Listing 5: Priority determination

4 Integration with Formal Verification Systems

4.1 ACL2 Integration

ACL2's theorem proving capabilities integrate naturally with our framework:

```
;; OpenCyc representation of ACL2
    theorem
(#$acl2Theorem #$SortingCorrectness
  (#$implies (#$true-listp ?x)
             (#$orderedp (#$sort ?x))))

;; Automatic theorem generation
(#$implies
  (#$needsProof ?function ?property)
  (#$generateACL2Theorem ?function ?
      property))
```

Listing 6: ACL2 integration

The agent can:

1. **Generate ACL2 definitions** from high-level specifications
2. **Attempt automated proofs** using ACL2's proof tactics
3. **Learn from proof failures** to improve code generation
4. **Maintain verification invariants** across code modifications

4.2 Lean4 Integration

Lean4's dependent type system provides rich specification capabilities:

```
;; OpenCyc representation of Lean4 types
(#$leanType #$SortedList
  (#$dependentType #$List
    (#$constraint #$Sorted)))

;; Automatic type inference
(#$implies
  (#$functionSignature ?f ?inputType ?
      outputType)
  (#$generateLeanType ?f (?inputType ->
      ?outputType)))
```

Listing 7: Lean4 integration

4.3 Verification-Driven Development

The agent follows a verification-first approach:

1. **Formal Specification**: Requirements encoded as logical formulas
2. **Implementation Synthesis**: Code generated to satisfy specifications
3. **Automated Verification**: Proofs attempted using theorem provers
4. **Refinement**: Failed verifications guide code improvements
5. **Knowledge Encoding**: Successful patterns stored for reuse

5 Implementation Case Study: Self-Improving OpenCyc

We demonstrate our framework through a concrete implementation: an autonomous agent that develops and improves its own OpenCyc knowledge base system.

5.1 Initial Knowledge Base

The agent begins with basic knowledge about:

- Prolog programming principles
- OpenCyc inference rules
- Software testing methodologies
- Common algorithmic patterns

5.2 Self-Assessment Capabilities

```
;; Agent queries its own capabilities
(#$currentCapability #$MySelf #
    $BasicInference)
(#$missingCapability #$MySelf #
    $CycLParsing)
(#$developmentPriority #$CycLParsing #
    $High)
```

Listing 8: Self-assessment

5.3 Autonomous Implementation

The agent autonomously:

1. **Identifies missing features** through knowledge base queries
2. **Plans implementation strategies** based on software engineering principles
3. **Generates code** using LLM integration with formal constraints
4. **Verifies correctness** through automated testing and theorem proving
5. **Integrates successful implementations** into its knowledge base

5.4 Learning and Improvement

Each development cycle enhances the agent's capabilities:

```
;; Learning from successful
    implementations
(#$implies
  (#$successfulImplementation ?pattern ?
      context)
  (#$preferredPattern ?pattern ?context)
      )

;; Improving development strategies
(#$implies
  (#$implementationFailed ?strategy ?
      reason)
  (#$avoidStrategy ?strategy ?
      similarContext))
```

Listing 9: Learning from experience

5.5 Results

Our prototype demonstrates:

- **Autonomous feature development**: Agent independently implements CycL parsing, advanced inference rules, and performance optimizations
- **Self-improving architecture**: Code quality and development velocity improve over time
- **Formal verification**: All implementations verified against formal specifications
- **Knowledge accumulation**: Development expertise encoded and reused across projects

6 Theoretical Foundations

6.1 Soundness Guarantees

Our framework provides several levels of correctness guarantees:

Logical Soundness: All inferences performed by the OpenCyc knowledge base are logically sound, ensuring that derived conclusions follow validly from premises.

Specification Compliance: Generated code is verified against formal specifications using theorem provers, providing mathematical guarantees of correctness.

Architectural Consistency: The knowledge base maintains invariants about software architecture, preventing the introduction of architectural violations.

Development Process Correctness: The autonomous development cycle follows verified

software engineering methodologies encoded in the knowledge base.

6.2 Completeness Considerations

While our framework cannot guarantee completeness (finding all possible solutions), it provides several completeness improvements over current approaches:

Systematic Exploration: The knowledge base guides the agent to explore solution spaces systematically rather than randomly.

Cumulative Learning: Previous solutions are retained and reused, preventing redundant exploration.

Formal Guidance: Theorem provers guide the search toward provably correct solutions.

6.3 Termination and Convergence

The autonomous development cycle is designed to converge toward optimal solutions:

Monotonic Improvement: Each successful implementation improves the system's capabilities without degrading existing functionality.

Bounded Search: Formal specifications bound the search space for solutions.

Progress Metrics: Quantitative measures ensure forward progress in each development cycle.

7 Comparison with Related Work

Table 1: Comparison with AI-Assisted Development Tools

Approach	Knowledge Persistence	Formal Reasoning	Autonomous Planning	Self-Improvement
GitHub Copilot	None	None	None	None
ChatGPT/Claude	Session-only	Informal	Limited	None
Our Framework	**Persistent KB**	**Formal Logic**	**Autonomous**	**Continuous**

Table 2: Comparison with Formal Verification Systems

System	Automation Level	Learning Capability	Dev Int
ACL2	Manual proofs	None	E
Lean4	Semi-automated	Limited	E
Coq	Manual proofs	None	E
Our Framework	**Fully Automated**	**Continuous**	**Int**

8 Applications and Use Cases

8.1 Autonomous System Development

Critical Systems: Autonomous agents can develop safety-critical software with formal verification guarantees, suitable for aerospace, medical devices, and autonomous vehicles.

Infrastructure Software: Self-improving compilers, operating systems, and database systems that optimize themselves based on usage patterns and performance data.

8.2 Research Acceleration

Theorem Proving: Agents that autonomously discover and prove mathematical theorems, accelerating research in logic and mathematics.

Algorithm Discovery: Systematic exploration of algorithmic design spaces with formal correctness guarantees.

8.3 Educational Applications

Intelligent Tutoring: Systems that understand student programming errors through formal analysis and provide targeted guidance.

Curriculum Development: Autonomous generation of programming exercises with verified solutions and difficulty progression.

9 Challenges and Future Work

9.1 Scalability Challenges

Knowledge Base Size: As the OpenCyc knowledge base grows, inference performance

may degrade. Future work should explore hierarchical knowledge organization, lazy evaluation strategies, and distributed reasoning systems.

Verification Complexity: Formal verification of large systems remains computationally challenging. Potential solutions include compositional verification techniques, approximate verification for non-critical components, and machine learning-guided proof search.

9.2 Integration Challenges

Tool Ecosystem: Integrating diverse verification tools (ACL2, Lean4, Coq) requires standardized specification languages, universal proof formats, and cross-system knowledge translation.

Human-Agent Collaboration: Balancing autonomous operation with human oversight requires explainable AI techniques, human-interpretable knowledge representations, and graceful degradation when human intervention is needed.

9.3 Future Research Directions

Multi-Agent Development: Teams of specialized agents collaborating on large software projects with architecture agents, implementation agents, testing agents, and performance agents.

Cross-Domain Knowledge Transfer: Applying development knowledge learned in one domain to another, including web development patterns applied to systems programming and algorithmic optimizations transferred between problem domains.

Evolutionary Software Architecture: Systems that autonomously evolve their architecture based on changing requirements through automatic refactoring, performance-driven transformations, and security-aware system hardening.

10 Conclusion

We have presented a novel framework for autonomous software development that combines formal knowledge representation, large language models, and machine verification systems. Our approach addresses fundamental limitations of current AI-assisted development tools by providing:

1. **Persistent Knowledge**: Formal encoding of software engineering principles and development history
2. **Autonomous Planning**: Self-directed identification of development priorities and implementation strategies
3. **Formal Verification**: Mathematical guarantees of software correctness through integrated theorem proving
4. **Continuous Learning**: Self-improvement through experience and knowledge accumulation

The framework demonstrates practical feasibility through our OpenCyc implementation case study, showing how an agent can autonomously develop and improve its own knowledge base system while maintaining formal correctness guarantees.

This work opens new research directions at the intersection of artificial intelligence, formal methods, and software engineering. The potential applications span from safety-critical systems development to research acceleration and educational tools.

Most significantly, our framework provides a path toward genuinely autonomous software engineering - systems that can understand, design, implement, and verify complex software systems with minimal human intervention while maintaining mathematical guarantees of correctness.

As AI systems become increasingly capable, the integration of formal knowledge representation and machine verification becomes essential for ensuring their reliability and safety. Our framework demonstrates that this integration is not only possible but practically achievable, paving the way for a new generation of autonomous development systems.

References

[1] Lenat, D. B. (1995). CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11), 33-38.

[2] Kaufmann, M., Manolios, P., & Moore, J. S. (2000). *Computer-aided reasoning: An approach*. Kluwer Academic Publishers.

[3] de Moura, L., Kong, S., Avigad, J., van Doorn, F., & von Raumer, J. (2015). The Lean theorem prover. *International Conference on Automated Deduction*, 378-388.

[4] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[5] Solar-Lezama, A. (2008). Program synthesis by sketching. UC Berkeley.

[6] Torlak, E., & Bodik, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6), 530-541.

[7] Gulwani, S., Polozov, O., & Singh, R. (2017). Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2), 1-119.

[8] Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., & Barrett, C. (2015). Counterexample-guided quantifier instantiation for synthesis in SMT. *International Conference on Computer Aided Verification*, 198-216.