Case study: solving P-99 with LPTP and an LLM

Fred Mesnard Thierry Marianne Étienne Payet
LIM, université de La Réunion, France
{frederic.mesnard,thierry.marianne,etienne.payet}@univ-reunion.fr

Wim Vanhoof
Université de Namur, Belgium
wim.vanhoof@unamur.be

Ninety-Nine Prolog Problems (P-99) is a famous set of Prolog exercises. We solved the first thirty three *just by prompting an LLM* (Large Language Model). We used Claude from Anthropic. By "solved" we mean: generate the Prolog code and a test file, run the tests and check whether they pass, then formally prove types, groundness, termination, uniqueness, existence and also sometimes functional correctness with LPTP (Logic Program Theorem Prover). Hence our approach is an experiment in *vibe-coding/vericoding* of P-99. It is a *vibe-coding* experiment because we started from informal specifications written in English and let Claude generate the Prolog code. It also fits within *vericoding* because the LLM proved *reliability guarantees* on the generated Prolog code. Claude wrote 58 logic procedures, 508 tests, 257 lemmas for a total of 11800 proof lines. We manually checked each file generated by the LLM. We checked the Prolog code, ran the tests, examined the logical statements generated by Claude and proof-checked Claude's proofs with LPTP. This paper describes this experiment and provides the main details so that it can be reproduced by the interested reader.

1 Introduction

Ninety-Nine Prolog Problems (P-99) is a famous set of Prolog exercises that have been translated to many other programming languages, including Erlang, Haskell, Lisp, Picat and Rust. There is also an Active Logic Document [15] for P-99 running Ciao-Prolog in any modern web browser[1]. The oldest copy of P-99 we have dates back to more than 25 years. These exercises were written by Werner Hett from the Bern University of Applied Sciences, Switzerland. The original site has been offline for many years. There are a few copies floating on the Internet (*e.g.*, here[2]). Some exercises are missing (P29, P30, P42-P45, P51-P53, P74-P79) and a few of them have extensions (P61A, P62B, P70B, P70C). All in all, there are 88 exercises.

In this work, we present a case study using the combination of an LLM and LPTP (Logic Program Theorem Prover) [21] to solve some of the exercises of P-99. We choose Claude from Anthropic and do the experiment in *Code mode* with the Opus 4.6 model released in February 2026. We solve the first thirty three exercises (P01-P28 + P31-P35 = 33/88 = 37.5%) just by *prompting Claude*. By "solve" we mean: generate the Prolog code and a test file, run the tests and check whether they pass, then formally prove types, groundness, termination, uniqueness, existence and also sometimes functional correctness with LPTP. In other words, our case study relates an experiment in *vibe-coding/vericoding* of P-99. It is a *vibe-coding* [18] experiment because we start from informal specifications written in English for students and let Claude generate the Prolog code. It also fits within a *vericoding* approach [4] because the LLM proved *reliability guarantees* on the generated Prolog code in a formal language that can be proof-checked.

[1] https://cliplab.org/logalg/doc/99problemsALD.html
[2] https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/

Preliminary Report. Final version to appear in:
ICLP 2026 Technical Communication

© F. Mesnard, T. Marianne, É. Payet, W. Vanhoof
This work is licensed under the
Creative Commons Attribution License.

We did check manually each of Claude's outputs. We checked the Prolog code, (re-)ran the tests, examined the logical statements generated by Claude and (re-)proof-checked Claude's proofs with LPTP. For functional correctness properties (informally: properties stating that the code computes what it is supposed to), we almost always provided some hints to Claude to express the logical statements we were interested in.

This paper describes this experiment and provides the main details so that it can be reproduced by the interested reader. It is organized as follows. Section 2 defines the experimental framework. Sections 3 and 4 describe how to instruct Claude about what it needs to know. Section 5 gives an overview of the experiment. Section 6 presents three examples with an emphasis on functional correctness properties. Section 7 describes an on-going work on implementing a Model Context Protocol connected to LPTP. Section 8 reviews some related work and Section 9 concludes.

2 Goal

The first problem of P-99 is given as follows:

```
P01 (*) Find the last element of a list.
Example:
?- my_last(X,[a,b,c,d]).
X = d
```

This is an informal specification written in English. In general, a specification includes at least one example query that indicates the predicate name, arity and expected usage. The `(*)` indicates the difficulty of the exercise, ranging from one to three stars.

We want the LLM to propose the corresponding Prolog code together with some tests. We want proven lemmas ensuring types, groundness, termination and functional properties (such as links with library predicates or previously solved exercises). We want to repeat this exercise for the next 32 problems.

More precisely, the programming language we instruct Claude to use is pure Prolog with equality `=/2` on finite trees, negation as failure, the if/then/else construct and *no* builtins. This subset is dictated by the input object language that LPTP can deal with (although the subset can be slightly enlarged for some builtins which can be logically approximated [21]). Natural numbers are represented using Peano notation (*e.g.*, 2 is noted as `s(s(0))`). The Prolog engine is assumed to run unification *with* occurs-check. SWI-Prolog has a special flag for this mode: `set_prolog_flag(occurs_check,true)`. Claude adds the corresponding directive at the beginning of each test file. The specification language and proof checker is LPTP [19, 20, 21], see also the companion paper [13] in this volume for a quick summary.

3 Claude setup

Let us download the GitHub repository[3] accompanying this paper. We get the `LPTP-LLM-P99-main` directory, which contains the files `CLAUDE.md` (see the next section), `P-99.html`, the P-99 Prolog problems in HTML format and the file `lptp-reference.md`. In this last file, Claude *itself* maintains tips and lessons learnt about the project, such as how to run LPTP and Prolog locally, common pitfalls and proof patterns.

[3] `https://github.com/FredMesnard/LPTP-LLM-P99`

For each Pxx from P99, we want Claude to create a directory P99/Pxx, with its own CLAUDE.md file describing the problem, its solution pxx.pl, its tests pxx_test.pl, the properties and their proofs pxx.pr, and a report pxx-experiment-report.md describing the work done. The file pxx.gr is a representation created by Claude of the Prolog file pxx.pl, to be used by LPTP. Table 1 displays the structure of the initial directory.

Table 1: Layout of the GitHub repository accompanying the paper

Directory	Content
P99/	CLAUDE.md, P-99.html, lptp-reference.md
P99/P01/	CLAUDE.md, p01-experiment-report.md, p01.gr, p01.pl, p01.pr, p01_test.pl

The next section is an exact copy of the contents of the file P99/CLAUDE.md. It contains all the instructions for Claude to solve a P-99 exercise. It has to be adapted to the local configuration. Once all the infrastructure is ready, the following prompt can be used for solving P02: *Read CLAUDE.md, lptp-reference.md, and solve the P02 exercise*.

4 CLAUDE.md

4.1 Project

Formal verification of the 99 Prolog Problems using LPTP (Logic Program Theorem Prover) v1.06. The goal is to provide Prolog code, a test file, and LPTP proofs of properties for some of the 99 Prolog Problems.

4.2 Reference

Read /Users/fred/Desktop/P99/lptp-reference.md before writing or debugging any .pr file. Update this reference if a new proof tip has been found.

4.3 Structure

The problems are listed in P-99.html. Each problem lives in a Pxx/ directory containing:

- pXX.pl — Prolog program
- pXX.gr — ground representation (variable-free clause encoding for LPTP)
- pXX.pr — proof file (lemmas, theorems)
- pXX_test.pl — SWI-Prolog test file (see Testing below)
- pXX-experiment-report.md — experiment report
- CLAUDE.md — problem specification

4.4 Conventions

- **Prolog** strictly ISO-compatible, = (equality, *i.e.*, finite-tree unification), \+ (negation as failure), and (... -> ... ; ...) (the if-then-else construct) are allowed, but no cut, no other built-in.

- Use predicates from the LPTP lib when available, as it will help for the proofs.
- Peano naturals: `0`, `s(0)`, `s(s(0))`, ... using the LPTP nat library.
- Hierarchical lemma names: `predicate:property` or `predicate:property:variant`.
- The predicates defined in `pXX.pl` and `pXX.gr` must correspond exactly.
- Copy `.gr` and `.pr` to `/Users/fred/lptp/tmp/` before verification.
- Command: `cd /Users/fred/lptp && printf "io__exec_file('tmp/pXX.pr').\nhalt.\n" | bin/lptp 2>&1`
- If `pXX.pr` uses lemmas from another problem (*e.g.*, P35 imports P31), declare the dependency via `:- needs_gr` / `:- needs_thm` and document it in the local `CLAUDE.md`.

4.5 Testing

For each problem Pxx, create a test file `pXX_test.pl` that:

- Starts with `:- set_prolog_flag(occurs_check,true).` as the very first line.
- Loads the program via `:- [pXX].`
- Includes helpers for Peano conversion (`to_peano/2`, `from_peano/2`, etc.) when needed.
- Tests each predicate defined in `pXX.pl` with representative cases: typical inputs, edge cases (empty list, 0, single element), and expected failures.
- Tests each semantic property proved in `pXX.pr` (types, uniqueness, ordered, product, etc.) as a runtime check.
- Prints `OK` or `FAIL` for each test case.
- Ends with `:- halt.`
- Runs with: `/Applications/SWI-Prolog.app/Contents/MacOS/swipl pXX_test.pl`

4.6 Properties to verify systematically

1. **Types** (`pred:types`) — type preservation (list, nat, etc.)
2. **Groundness** (`pred:ground`) — ground inputs $\Rightarrow$ ground outputs
3. **Termination** (`pred:termination`) — termination under preconditions
4. **Uniqueness** (`pred:uniqueness`) — determinism of the result
5. **Existence** (`pred:existence`) — existence of the result
6. **Functional Correctness** — any link with library predicates or previously solved exercises

4.7 Experiment Report

Each `pXX-experiment-report.md` should contain:

- **Problem statement** — what the predicate does
- **Prolog code** — summary of the predicates defined
- **Properties proved** — list of lemmas with their statement and proof technique (completion, structural induction, strengthened induction, algebraic, etc.)
- **Statistics** — line counts (`.pl`, `.gr`, `.pr`), number of lemmas, proof-to-code ratio
- **Difficulties and lessons learned** — LPTP pitfalls encountered, proof strategies that worked

4.8 Language

- Code, lemma names, and LPTP proofs: English.
- Experiment reports and documentation: English.
- Conversation with the user: French (the user's preferred language).

5 Overview of the experiment

Together with Claude, we solved the first 33 Prolog exercises. Understanding the specification, writing the Prolog code and the test file takes a few minutes. We noticed that from time to time, a *context compaction* is followed by *this session is being continued from a previous conversation that ran out of context*. It can be useful to tell Claude to reread the `CLAUDE.md` files as Claude may have forgotten some guidelines. Switching to the verification part, the process is much slower. For functional properties, we had to give hints to Claude. For instance, for P01 and P02, we asked: *what is the connection with append/3?* Sometimes we had to fully formulate the properties in natural language (see Section 6.3). Solving one exercise takes Claude between 15 minutes (*e.g.*, P01) to several hours (*e.g.*, P35). We checked manually each generated file. We checked the Prolog code, we ran the tests, we examined the logical statements and we checked the proofs with LPTP.

During this experiment, Claude created 58 logic procedures, with a total of 112 Prolog clauses, 150 lines of code and 508 runtime tests. It did make use of negation as failure but made no use of the if/then/else construct. Although P-99 is a well-known Prolog resource, as we explicitly asked for pure Prolog code, the code generated by Claude is often quite different from the Prolog solutions one may find on the Internet (see *e.g.*, Section 6.2 and Section 6.3), which very frequently include impure constructs. Claude proved 257 lemmas and wrote about 11800 lines of proof. It had no problem in using what is already available in the LPTP library, code and lemmas. While proving its lemmas, it found various proof techniques which it added and documented in the `lptp-reference.md` file. An example of such a proof technique is the following: sometimes we have to strengthen the property to be proved so that we can do the inductive proof, and then weaken the property to get the original one. The human readable `lptp-reference.md` file is maintained by Claude and is included in the GitHub repository accompanying this paper to speed up another run of this experiment.

6 Selected examples

6.1 P01: the last element of a list

The problem statement of exercise P01 was given in Section 2. Claude produces the following Prolog code (naive solution):

```
my_last(X, [X]).
my_last(X, [_|L]) :- my_last(X, L).
```

It also infers and proves 10 properties, see Table 2. We simplify the LPTP syntax for readability. The full statements and their LPTP-checked proofs are listed in the `p01.pr` file. The `p01_test.pl` file contains 21 tests, both for the source code and for the first 7 properties. The last three properties are Claude's answer to our question: *what is the connection with append/3?* Actually, properties 8 and 10 constitute a characterization of `my_last/2` w.r.t. `append/3`.

Table 2: Properties proved for `my_last/2`

#	Property	Statement
1	Termination	$\forall x,l.\ \texttt{list}(l) \Rightarrow \textbf{terminates}\ \texttt{my\_last}(x,l)$
2	Type preservation	$\forall x,l.\ \texttt{my\_last}(x,l) \Rightarrow \texttt{list}(l)$
3	Groundness	$\forall x,l.\ \texttt{my\_last}(x,l) \wedge \textbf{gr}(l) \Rightarrow \textbf{gr}(x)$
4	Membership	$\forall x,l.\ \texttt{my\_last}(x,l) \Rightarrow \texttt{member}(x,l)$
5	Uniqueness	$\forall x,l.\ \texttt{my\_last}(x,l) \Rightarrow (\forall y.\ \texttt{my\_last}(y,l) \Rightarrow x = y)$
6	Existence (aux.)	$\forall z,l.\ \texttt{list}(l) \Rightarrow \exists x.\ \texttt{my\_last}(x,[z\|l])$
7	Existence	$\forall l.\ \texttt{list}(l) \wedge l \neq [] \Rightarrow \exists x.\ \texttt{my\_last}(x,l)$
8	Forward append	$\forall x,l.\ \texttt{my\_last}(x,l) \Rightarrow \exists l_1.\ \texttt{append}(l_1,[x],l)$
9	Converse append (aux.)	$\forall x,l_1,l.\ \texttt{list}(l_1) \wedge \texttt{append}(l_1,[x],l) \Rightarrow \texttt{my\_last}(x,l)$
10	Converse append	$\forall x,l_1,l.\ \texttt{append}(l_1,[x],l) \Rightarrow \texttt{my\_last}(x,l)$

6.2 P31: a primality test

P31 asks for a primality test.

```
P31 (**) Determine whether a given integer number is prime.
Example:
?- is_prime(7).
Yes
```

Claude generates the following Prolog code:

```
% divides(D, N) - D divides N (by repeated subtraction, D > 0).
divides(D, D).
divides(D, N) :- plus(D, M, N), divides(D, M).

% no_factor(K, D, N) - none of D, D+1, ..., D+K-1 divides N.
%  K is a countdown: the number of candidates left to check.
no_factor(0, _, _).
no_factor(s(K), D, N) :- \+ divides(D, N), no_factor(K, s(D), N).

% is_prime(N) - N is prime (N >= 2 and no factor from 2 to N-1).
%  For N = s(s(X)), check X candidates starting from 2.
is_prime(s(s(X))) :- no_factor(X, s(s(0)), s(s(X))).
```

In Table 3, we extract the two properties[4] stating soundness and completeness of the implementation. We did ask for such properties and gave the idea of their formalization in natural language. Then Claude was able to formalize and prove them.

6.3 P35: decomposition in prime numbers

P35 asks for a decomposition of a natural number into prime factors.

```
P35 (**) Determine the prime factors of a given positive integer.
Construct a flat list containing the prime factors in ascending order.
Example:
```

[4] Numbering as in the `p31.pr` file.

Table 3: Soundness and completeness for `is_prime/1`

#	Property	Statement
7	Soundness of `is_prime`	$\forall n.\, \texttt{nat}(n) \wedge \texttt{is\_prime}(n) \Rightarrow$ $\exists x.\, n = s(s(x)) \wedge (\forall d_1.\, s(0) \leq d_1 \wedge d_1 < s(x) \Rightarrow$ **fails** $\texttt{divides}(s(d_1), n))$
9	Completeness of `is_prime`	$\forall n, x.\, \texttt{nat}(n) \wedge n = s(s(x)) \wedge$ $(\forall d_1.\, s(0) \leq d_1 \wedge d_1 < s(x) \Rightarrow$ **fails** $\texttt{divides}(s(d_1), n)) \Rightarrow \texttt{is\_prime}(n)$

```
?- prime_factors(315, L).
L = [3,3,5,7]
```

Here is the Prolog code generated by Claude:

```
% quot(D, N, Q) - quotient Q = N/D (assumes D divides N, D > 0).
quot(D, D, s(0)).
quot(D, N, s(Q)) :- plus(D, M, N), quot(D, M, Q).

% smallest_factor(N, D, K, F) - smallest factor F of N starting from
%  candidate D, with countdown K for termination.
%  K counts how many candidates remain to try after D.
smallest_factor(N, D, K, D) :- divides(D, N).
smallest_factor(N, D, s(K), F) :- \+ divides(D, N),
  smallest_factor(N, s(D), K, F).

% prime_factors(N, L) - L is the list of prime factors of N (>= 1)
%  in ascending order, with repetitions.
%  N = 0 fails (not a positive integer).
prime_factors(s(0), []).
prime_factors(s(s(X)), [F|L]) :-
  smallest_factor(s(s(X)), s(s(0)), X, F),
  quot(F, s(s(X)), Q),
  prime_factors(Q, L).
```

Table 4 lists the properties of this program. Claude started with termination and type properties. For stating and proving properties 23 to 30 that we explicitly asked in natural language, Claude reused code and lemmas from the `nat` library (*e.g.*, `nat/1`, `plus/3`, `times/3`, `@=< /2`), derived and proved new lemmas for P31 (`divides/2`), P35 (`quot/3`, `smallest_factor/4`, `prime_factors/2`) and added Prolog code (`ordered/1`, `product/2`):

```
% ordered(L) - L is sorted in ascending order (using @=<).
ordered([]).
ordered([_]).
ordered([X,Y|L]) :- X @=< Y, ordered([Y|L]).

% product(L, P) - P is the product of elements of L.
product([], s(0)).
product([X|L], P) :- product(L, P1), times(X, P1, P).
```

We note that the product of the natural numbers contained in the empty list is 1. Let us focus on the last few properties. Assuming n is a Peano integer and $prime\_factors(n,l)$ succeeds, here is what Claude proved:

- property 23 (or 24): the resulting list l is a list of Peano integers;
- property 26: the product of the elements of l is n;
- property 27: l is in ascending order;
- property 28: each element of l is a prime number;
- property 30: l is unique.

We also get a sufficient condition for the existence of a solution:

- property 29: if n is strictly greater than 0, there exists l such that $prime\_factors(n,l)$ succeeds.

Actually the condition is also necessary as $prime\_factors(0,\_)$ finitely fails. All together, these results constitute a logic-programming-based proof of the *prime factorization theorem*: "every integer greater than 1 is either prime or can be represented uniquely as a product of prime numbers, up to the order of the factors"[5]. Note that the Prolog code states that the prime factors of 1 is the empty list, which is meaningful. So on the one hand, the Prolog code effectively constructs the ordered list l of prime factors of a strictly positive natural number n. On the other hand, the proofs certify that if n is strictly greater than 0 then l always exists, is unique and correct.

7 Proof generation and certification using an MCP

7.1 MCP server tools

In November 2024, Anthropic released the Model Context Protocol (MCP) [2]. Relying on this open-source standard, we have implemented the `atp-lptp-mcp` server[6] by using the TypeScript SDK. It allows conversational assistants and agentic tools such as Claude Code to use LPTP directly. Concretely, the MCP server exposes LPTP-tailored tools to assistants like Claude Code and Gemini. The generative AI agent acts as an MCP client; it generates proof terms that are submitted to the MCP server and thus checked by LPTP.

As such, using an MCP server is a viable alternative to providing the `lptp-reference.md` file prior to generating the proof terms. Using the MCP server offers several advantages. First, LPTP's ISO-Prolog syntax is embedded within the server via the `get_lptp_grammar` tool. Furthermore, as we no longer need to provide assistant-specific documentation in Markdown format or the LPTP reference, using MCP reduces token consumption during each interaction. Finally, we can easily reuse the same MCP configuration with different LLM-based assistants and compare the outputs.

7.2 Interactive Theorem Proving and LLM-based assistants

Our MCP server maps calls to LPTP tactics such as induction, case analysis, completion, definition unfolding, existential formula elimination, the totality axiom expansion or automated proof search. These tactics are applied until a complete derivation is generated. We opted for mapping each native tactic to a separate tool so that we could request the LLM assistants for proof sketches containing tactic sequences, before realizing that this intermediate step was not required. The final architecture embeds the most

[5] `https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic`

[6] `https://www.npmjs.com/package/atp-lptp-mcp`

#	Property	Statement
quot/3 — quotient		
1	Termination	$\forall d_0,n,q.\ \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \Rightarrow$ **terminates** $\mathtt{quot}(s(d_0),n,q)$
2	Types	$\forall d_0,n,q.\ \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{quot}(s(d_0),n,q) \Rightarrow \mathtt{nat}(q)$
3	Strict bound	$\forall d_1,n,q.\ \mathtt{nat}(d_1) \wedge \mathtt{nat}(n) \wedge \mathtt{quot}(s(s(d_1)),n,q) \Rightarrow q < n$
4	Correctness	$\forall d_0,n,q.\ \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{quot}(s(d_0),n,q) \Rightarrow s(d_0) \times q = n$
5	Positive	$\forall d,n,q.\ \mathtt{quot}(d,n,q) \Rightarrow \exists q_0.\ q = s(q_0)$
6	Success	$\forall d,n.\ \mathtt{nat}(d) \wedge \mathtt{nat}(n) \wedge \mathtt{divides}(d,n) \Rightarrow \exists q.\ \mathtt{quot}(d,n,q)$
7	Uniqueness	$\forall d_0,n,q_1,q_2.\ \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{quot}(s(d_0),n,q_1) \wedge \mathtt{quot}(s(d_0),n,q_2) \Rightarrow q_1 = q_2$
smallest_factor/4		
8	Termination	$\forall k,d_0,n,f.\ \mathtt{nat}(k) \wedge \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \Rightarrow$ **terminates** $\mathtt{smallest\_factor}(n,s(d_0),k,f)$
9	Types	$\forall k,d_0,n,f.\ \mathtt{nat}(k) \wedge \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{smallest\_factor}(n,s(d_0),k,f) \Rightarrow \mathtt{nat}(f)$
10	Lower bound	$\forall k,d_0,n,f.\ \mathtt{nat}(k) \wedge \mathtt{smallest\_factor}(n,s(d_0),k,f) \Rightarrow \exists f_0.\ f = s(f_0)$
11	Lower bound 2	$\forall k,d_0,n,f.\ \mathtt{nat}(k) \wedge \mathtt{smallest\_factor}(n,s(s(d_0)),k,f) \Rightarrow \exists f_1.\ f = s(s(f_1))$
12	Divides	$\forall k,d_0,n,f.\ \mathtt{nat}(k) \wedge \mathtt{smallest\_factor}(n,s(d_0),k,f) \Rightarrow \mathtt{divides}(f,n)$
13	Completeness	$\forall k,d_0,n,last.\ \mathtt{nat}(k) \wedge \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{plus}(d_0,k,last) \wedge \mathtt{divides}(s(last),n) \Rightarrow \exists f.\ \mathtt{smallest\_factor}(n,s(d_0),k,f)$
14	Leq factor	$\forall k,d_0,n,f,g.\ \mathtt{nat}(k) \wedge \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{smallest\_factor}(n,s(d_0),k,f) \wedge \mathtt{divides}(g,n) \wedge s(d_0) \leq g \wedge \mathtt{nat}(g) \Rightarrow f \leq g$
15	Uniqueness	$\forall k,d_0,n,f_1,f_2.\ \mathtt{nat}(k) \wedge \mathtt{nat}(d_0) \wedge \mathtt{nat}(n) \wedge \mathtt{smallest\_factor}(n,s(d_0),k,f_1) \wedge \mathtt{smallest\_factor}(n,s(d_0),k,f_2) \Rightarrow f_1 = f_2$
16	Is prime	$\forall k,n,f.\ \mathtt{nat}(k) \wedge \mathtt{nat}(n) \wedge \mathtt{smallest\_factor}(n,s(s(0)),k,f) \Rightarrow \mathtt{is\_prime}(f)$
divides/2 — auxiliary		
17	Self	$\forall d.\ \mathtt{divides}(d,d)$
18	Sum	$\forall d,n,b,c.\ \mathtt{divides}(d,n) \wedge \mathtt{divides}(d,b) \wedge \mathtt{nat}(n) \wedge \mathtt{nat}(b) \wedge \mathtt{nat}(d) \wedge \mathtt{plus}(n,b,c) \Rightarrow \mathtt{divides}(d,c)$
19	Times factor	$\forall f_0,g,q.\ \mathtt{nat}(f_0) \wedge \mathtt{nat}(q) \wedge \mathtt{nat}(g) \wedge \mathtt{divides}(g,q) \Rightarrow \mathtt{divides}(g,s(f_0) \times q)$
20	Transitive	$\forall a,b,c.\ \mathtt{nat}(a) \wedge \mathtt{nat}(b) \wedge \mathtt{nat}(c) \wedge \mathtt{divides}(a,b) \wedge \mathtt{divides}(b,c) \Rightarrow \mathtt{divides}(a,c)$
prime_factors/2		
21	Termination	$\forall n,l.\ \mathtt{nat}(n) \Rightarrow$ **terminates** $\mathtt{prime\_factors}(s(n),l)$
22	Types	$\forall n,l.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l) \Rightarrow \mathtt{list}(l)$
23	Nat members	$\forall n,l.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l) \Rightarrow (\forall z.\ \mathtt{member}(z,l) \Rightarrow \mathtt{nat}(z))$
24	Nat list	$\forall n,l.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l) \Rightarrow \mathtt{nat\_list}(l)$
25	Head info	$\forall n,h,t.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,[h\|t]) \Rightarrow \mathtt{divides}(h,n) \wedge \mathtt{nat}(h) \wedge (\exists f_1.\ h = s(s(f_1)))$
26	Product	$\forall n,l.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l) \Rightarrow \mathtt{product}(l,n)$
27	Ordered	$\forall n,l.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l) \Rightarrow \mathtt{ordered}(l)$
28	All prime	$\forall n,l,z.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l) \wedge \mathtt{member}(z,l) \Rightarrow \mathtt{is\_prime}(z)$
29	Existence	$\forall n.\ \mathtt{nat}(n) \Rightarrow \exists l.\ \mathtt{prime\_factors}(s(n),l)$
30	Uniqueness	$\forall n,l_1,l_2.\ \mathtt{nat}(n) \wedge \mathtt{prime\_factors}(n,l_1) \wedge \mathtt{prime\_factors}(n,l_2) \Rightarrow l_1 = l_2$

Table 4: Properties proved for P35

general `apply_tactic` tool to apply arbitrary tactics depending on the target proof. Regular proof checking is requested to LPTP via dedicated MCP tools.

We instructed both Gemini 3.1 Pro and Claude Code (with its cheaper Sonnet 4.6 model) to synthesize LPTP properties (proofs with gaps) from logic programs corresponding to P14-P24 in P-99. This subset is sufficiently diverse to illustrate properties such as termination, determinism, groundness, type, existence and uniqueness. We noticed that Gemini generates a more diverse set of properties than Claude. The LLMs were then instructed to fill the gaps. We observed that only Claude generated valid proofs for the P14-P24 subset.

The MCP implementation can verify individual proof steps (by applying `verify_lemma` to a single proof term) as well as entire proof files (`check_proof`). It runs LPTP on top of SWI-Prolog 10.0.2. The MCP server converts the Prolog interpreter's answers to queries into JSON structures, again consumed by the LLM assistant. The LLM assistant keeps revising and submitting the produced derivations until they are certified by LPTP. This architecture ensures there are only two possible states at the end of the process: proofs with gaps or formally certified proofs. All property batches were certified within 40 minutes to one hour of interaction depending on the batch complexity. We issued for each batch a single instruction to close all the gaps by only using the MCP LPTP tools. Table 5 lists properties generated and certified for the main predicates.

8 Related work

The recent arrival and rapid adoption of LLM's and their growing aptitude at writing and/or dealing with code has triggered different AI assisted programming practices or methodologies. We can distinguish the following, even though in practice they tend to overlap and hybrid approaches exist [14, 17, 24].

Among these methodologies, *vibe-coding* [11, 18] is presumably the most widespread and most easily accessible technique, for novices as well as experienced professionals. Vibe-coding is mostly understood as the process in which the user provides the LLM with a prompt representing an informal specification of a problem written in natural language, the LLM produces a solution in code, and the user tests and ideally accepts (possibly after multiple iterations) the given solution [7, 18]. While the burden of validation lies with the user, preliminary reviews in the literature argue that testing is often skipped or even delegated to the AI [7]. Consequently, benchmarks show that code generated by vibe-coding often exhibits functional correctness issues and sometimes raises serious security concerns [7, 27].

A natural approach to alleviate some of the issues presented by vibe-coding is to augment the process with formal specification and verification (see, a.o., [14, 24]). In what is sometimes called *vericoding* [4], the user provides a *formal* specification of the problem upfront, thereby limiting accessibility of the technique to a more expert audience. The LLM then generates a verifiable implementation of the specification, i.e. it accompanies the generated code with a machine-checkable proof of its correctness. While the use of formal verification can in principle eliminate the functional correctness issues sometimes encountered in vibe-coding [4, 14], correctness (and other) guarantees are obviously highly dependent on the quality and completeness of the specification. Available benchmarks [4, 26] show results depending on the formal language used. In these benchmarks, using Dafny seems to perform better than Verus and Lean [4, 26], but the effective success rates vary widely and, as the authors note, are subject to the rapid progress of the LLM models themselves. While these results are encouraging, other results are less optimistic. For example, in [22], the authors report very low success rates when evaluating a more demanding benchmark in which both the specification and the implementation are required to be generated and verified via Lean.

Program	**Properties**
P14 (`dupli/2`)	$\forall l,m : \text{list}(l) \wedge \text{dupli}(l,m) \Rightarrow \text{list}(m)$ $\forall l,m : \text{gr}(l) \wedge \text{dupli}(l,m) \Rightarrow \text{gr}(m)$ $\forall l,m : \text{list}(l) \Rightarrow$ **terminates** $\text{dupli}(l,m)$ $\forall l,m_1,m_2 : \text{dupli}(l,m_1) \wedge \text{dupli}(l,m_2) \Rightarrow m_1 = m_2$
P15 (`dupli/3`)	$\forall l,n,m : \text{list}(l) \wedge \text{nat}(n) \wedge \text{dupli}(l,n,m) \Rightarrow \text{list}(m)$ $\forall l,n,m : \text{gr}(l) \wedge \text{gr}(n) \wedge \text{dupli}(l,n,m) \Rightarrow \text{gr}(m)$ $\forall l,n,m : \text{list}(l) \wedge \text{nat}(n) \Rightarrow$ **terminates** $\text{dupli}(l,n,m)$ $\forall l,n,m_1,m_2 : \text{dupli}(l,n,m_1) \wedge \text{dupli}(l,n,m_2) \Rightarrow m_1 = m_2$
P16 (`drop/3`)	$\forall l,n,m : \text{list}(l) \wedge \text{nat}(n) \wedge \text{drop}(l,n,m) \Rightarrow \text{list}(m)$ $\forall l,n,m : \text{gr}(l) \wedge \text{nat}(n) \wedge \text{drop}(l,n,m) \Rightarrow \text{gr}(m)$ $\forall l,n,m : \text{list}(l) \wedge \text{nat}(n) \Rightarrow$ **terminates** $\text{drop}(l,n,m)$ $\forall l,n,m_1,m_2 : \text{list}(l) \wedge \text{nat}(n) \wedge \text{drop}(l,n,m_1) \wedge \text{drop}(l,n,m_2) \Rightarrow m_1 = m_2$
P17 (`split/4`)	$\forall l,n,l_1,l_2 : \text{list}(l) \wedge \text{nat}(n) \wedge \text{split}(l,n,l_1,l_2) \Rightarrow \text{list}(l_1) \wedge \text{list}(l_2)$ $\forall l,n,l_1,l_2 : \text{gr}(l) \wedge \text{split}(l,n,l_1,l_2) \Rightarrow \text{gr}(l_1) \wedge \text{gr}(l_2)$ $\forall l,n,l_1,l_2,l_3,l_4 : \text{split}(l,n,l_1,l_2) \wedge \text{split}(l,n,l_3,l_4) \Rightarrow l_1 = l_3 \wedge l_2 = l_4$
P18 (`slice/4`)	$\forall l,i,k,r : \text{list}(l) \wedge \text{nat}(i) \wedge \text{nat}(k) \wedge \text{slice}(l,i,k,r) \Rightarrow \text{list}(r)$ $\forall l,i,k,r : \text{gr}(l) \wedge \text{gr}(i) \wedge \text{gr}(k) \wedge \text{slice}(l,i,k,r) \Rightarrow \text{gr}(r)$ $\forall l,i,k,r : \text{list}(l) \wedge \text{nat}(i) \wedge \text{nat}(k) \Rightarrow$ **terminates** $\text{slice}(l,i,k,r)$ $\forall l,i,k,r_1,r_2 : \text{list}(l) \wedge \text{nat}(i) \wedge \text{nat}(k) \wedge \text{slice}(l,i,k,r_1) \wedge \text{slice}(l,i,k,r_2) \Rightarrow r_1 = r_2$
P19 (`rotate/3`)	$\forall x,n,y : \text{list}(x) \wedge \text{nat}(n) \wedge \text{rotate}(x,n,y) \Rightarrow \text{list}(y)$ $\forall x,n,y : \text{gr}(x) \wedge \text{gr}(n) \wedge \text{rotate}(x,n,y) \Rightarrow \text{gr}(y)$ $\forall l,n,r_1,r_2 : \text{list}(l) \wedge \text{nat}(n) \wedge \text{rotate}(l,n,r_1) \wedge \text{rotate}(l,n,r_2) \Rightarrow r_1 = r_2$
P20 (`remove_at/4`)	$\forall x,l,n,r : \text{list}(l) \wedge \text{nat}(n) \wedge \text{remove\_at}(x,l,n,r) \Rightarrow \text{list}(r)$ $\forall x,l,n,r : \text{gr}(x) \wedge \text{gr}(l) \wedge \text{gr}(n) \wedge \text{remove\_at}(x,l,n,r) \Rightarrow \text{gr}(r)$ $\forall x_1,x_2,l,n,r_1,r_2 : \text{list}(l) \wedge \text{nat}(n) \wedge \text{remove\_at}(x_1,l,n,r_1) \wedge$ $\text{remove\_at}(x_2,l,n,r_2) \Rightarrow x_1 = x_2 \wedge r_1 = r_2$
P21 (`insert_at/4`)	$\forall x,l,n,r : \text{list}(l) \wedge \text{nat}(n) \wedge \text{insert\_at}(x,l,n,r) \Rightarrow \text{list}(r)$ $\forall x,l,n,r : \text{gr}(x) \wedge \text{gr}(l) \wedge \text{gr}(n) \wedge \text{insert\_at}(x,l,n,r) \Rightarrow \text{gr}(r)$ $\forall x,l,n,r_1,r_2 : \text{list}(l) \wedge \text{nat}(n) \wedge \text{insert\_at}(x,l,n,r_1) \wedge \text{insert\_at}(x,l,n,r_2) \Rightarrow r_1 = r_2$
P22 (`range/3`)	$\forall a,b,r : \text{nat}(a) \wedge \text{nat}(b) \wedge \text{range}(a,b,r) \Rightarrow \text{list}(r)$ $\forall a,b,r : \text{gr}(a) \wedge \text{gr}(b) \wedge \text{range}(a,b,r) \Rightarrow \text{gr}(r)$ $\forall a,b,r_1,r_2 : \text{nat}(a) \wedge \text{nat}(b) \wedge \text{range}(a,b,r_1) \wedge \text{range}(a,b,r_2) \Rightarrow r_1 = r_2$
P23 (`rnd_select/3`)	$\forall l,n,r : \text{list}(l) \wedge \text{nat}(n) \wedge \text{rnd\_select}(l,n,r) \Rightarrow \text{list}(r)$ $\forall l,n,r : \text{gr}(l) \wedge \text{gr}(n) \wedge \text{rnd\_select}(l,n,r) \Rightarrow \text{gr}(r)$
P24 (`lotto/3`)	$\forall n,m,r : \text{nat}(n) \wedge \text{nat}(m) \wedge \text{lotto}(n,m,r) \Rightarrow \text{list}(r)$ $\forall n,m,r : \text{gr}(n) \wedge \text{gr}(m) \wedge \text{lotto}(n,m,r) \Rightarrow \text{gr}(r)$

Table 5: Properties generated and certified via MCP for P14-P24

While in vibe-coding and vericoding a human user orchestrates the code (and, in the case of vericoding, the proof) generation, *agentic coding* [8, 23] goes further in the sense that the human user formulates a higher-level goal, and delegates to an autonomous AI agent a large part of the software production process (coding, developing and running tests, fixing bugs, and up to submitting pull requests) without human intervention. Verification is mostly empirical. While some speedups are reported, these appear highly context and task dependent [1, 3]. Moreover, the use of agents also appears to induce persistent quality risks. A recent analysis [25] reports that roughly half of agentic pull requests (for code passing automated tests) were rejected by the maintainers of a repository.

9 Conclusion

In this paper, we report on an experiment in which the first third of the well-known P-99 set of Prolog exercises is solved using LPTP in combination with an LLM. We used Claude (Anthropic) in Code mode with the Opus 4.6 model. By solving an exercise, we mean reading the natural language specification, providing the Prolog code, constructing and running a test file, and stating and proving the usual properties of the Prolog code (types, groundness, termination, existence, uniqueness). For functional properties, we interacted with the LLM and checked each output. The time needed to solve an exercise varies from 15 minutes (P01) to several hours (P35). We note that the code and proofs of P35 define an LP (logic programming) view of the *prime factorization theorem* of arithmetic. The final part of the paper presents ongoing work aimed at supporting lower-level interaction through the Model Context Protocol, enabling connections to any LLM that supports the protocol (*e.g.*, Gemini 3.1 Pro).

The *vibe-coding* part of the experiment—reading the specification, and writing the Prolog code and tests—was handled easily by the LLM. We first checked these outputs to ensure that we were working with the correct code, free from impure constructs such as the cut and from the use of Prolog's native numbers.

The *vericoding* part of the experiment—stating and proving reliability guarantees of the generated code—was clearly more challenging for the LLM; however, it was generally able to handle it, except for functional properties. One should keep in mind that the properties proved by the LLM have been the subject of decades of research work in program verification and abstract interpretation [5], in particular in the LP setting [6]. We have now many theoretical results and some implementations (*e.g.*, [9, 16]) which compute more precise results than those inferred by the LLM. We have already begun investigating the use of automated theorem provers to verify properties of Prolog programs expressed in LPTP [12], as well as the automatic generation of LPTP proofs for groundness properties [10].

Nonetheless, the ease and rapidity with which the LLM was able to grasp the (largely unknown) LPTP formalism are impressive. We even observed that, when faced with a difficult proof, Claude examined the Prolog source code of the proof checker to gain a deeper "understanding" of the proof-checking process and fix the problematic proof. Also combining an LLM with a proof checker avoids the "hallucinations" that the LLM may produce and allows it to fix its errors. Thus, the LPTP/LLM combination appears to be a valuable tool for the LP developer. This applies both to constructing LPTP proofs of invariants inferred by abstract interpretation—when no purely algorithmic method is available—and to proving functional properties of Prolog code, where the corresponding abstract domain is typically not implemented as it depends on the specific problem (see Section 6). An interesting problem to investigate is the *scalability* of this approach.

Acknowledgement. We thank Romain Pabot for enlightening discussions about LLMs.

References

[1] Shyam Agarwal, Hao He & Bogdan Vasilescu (2026): *AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development*. arXiv:2601.13597.

[2] Anthropic (2024): *Introducing the Model Context Protocol*. Anthropic Blog. `https://www.anthropic.com/news/model-context-protocol`.

[3] Joel Becker, Nate Rush, Elizabeth Barnes & David Rein (2025): *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity*. arXiv:2507.09089.

[4] Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan & Max Tegmark (2025): *A benchmark for vericoding: formally verified program synthesis*. CoRR abs/2509.22908, doi:10.48550/ARXIV.2509.22908. arXiv:2509.22908.

[5] P. Cousot & R. Cousot (1977): *Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints*. In: Proc. of the 4th Symp. on Principles of Programming Languages, ACM, pp. 238–252.

[6] P. Cousot & R. Cousot (1992): *Abstract interpretation and application to logic programs*. Journal of Logic Programming 13(2,3), pp. 103–179.

[7] Ahmed Fawzy, Amjed Tahir & Kelly Blincoe (2025): *Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – a Grey Literature Review*. In: Proceedings of the 48th International Conference on Software Engineering (ICSE 2026), Software Engineering in Practice (SEIP). Available at `https://arxiv.org/abs/2510.00328`.

[8] Ahmed E. Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa & Dong Qiu (2025): *Agentic Software Engineering: Foundational Pillars and a Research Roadmap*. CoRR abs/2509.06216. Available at `https://doi.org/10.48550/arXiv.2509.06216`.

[9] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno & Pedro López-García (2005): *Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor)*. Sci. Comput. Program. 58(1-2), pp. 115–140, doi:10.1016/J.SCICO.2005.02.006.

[10] Thierry Marianne, Fred Mesnard & Étienne Payet (2025): *Automated Certification of Logic Program Groundness Analysis*. In Santiago Escobar & Laura Titolo, editors: Logic-Based Program Synthesis and Transformation - 35th International Symposium, LOPSTR 2025, Rende, Italy, September 9-10, 2025, Proceedings, Lecture Notes in Computer Science, Springer, pp. 113–122, doi:10.1007/978-3-032-04848-6_7.

[11] Christian Meske, Tobias Hermanns, Esther Von der Weiden, Kai-Uwe Loser & Thorsten Berger (2025): *Vibe Coding as a Reconfiguration of Intent Mediation in Software Development: Definition, Implications, and Research Agenda*. IEEE Access 13, pp. 213242–213259. Available at `https://api.semanticscholar.org/CorpusID:280338092`.

[12] Fred Mesnard, Thierry Marianne & Étienne Payet (2026): *Automated Theorem Proving for Prolog Verification*. Electronic Proceedings in Theoretical Computer Science 439, p. 469–481, doi:10.4204/eptcs.439.32.

[13] Fred Mesnard, Étienne Payet & Wim Vanhoof (2026): *Case study: proving $\sqrt{2}$ irrational with LPTP and an LLM*. In Wolfgang Faber & Laura Giordano, editors: Proceedings of the 42nd International Conference on Logic Programming (Technical Communications), ICLP 2026, Electronic Proceedings in Theoretical Computer Science (EPTCS) , Open Publishing Association.

[14] Jacqueline Mitchell & Yasser Shaaban (2025): *Position: Vibe Coding Needs Vibe Reasoning: Improving Vibe Coding with Formal Verification*. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages, LMPL '25, Association for Computing Machinery, New York, NY, USA, p. 84–90, doi:10.1145/3759425.3763390.

[15] José F. Morales, Salvador Abreu, Daniela Ferreiro & Manuel V. Hermenegildo (2023): *Teaching Prolog with Active Logic Documents*. In David Scott Warren, Verónica Dahl, Thomas Eiter, Manuel V. Hermenegildo,

Robert A. Kowalski & Francesca Rossi, editors: *Prolog: The Next 50 Years*, Lecture Notes in Computer Science, Springer, pp. 171–183, doi:10.1007/978-3-031-35254-6_14.

[16] Étienne Payet & Fred Mesnard (2006): *Nontermination inference of logic programs*. *ACM Transactions on Programming Languages and Systems* 28(2), pp. 256–289, doi:10.1145/1119479.1119481.

[17] Ranjan Sapkota, Konstantinos I. Roumeliotis & Manoj Karkee (2025): *Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI*. arXiv:2505.19443.

[18] Advait Sarkar & Andrew Drosos (2025): *Vibe coding: programming through conversation with artificial intelligence*. In: *Proceedings of the 36th Annual Conference of the Psychology of Programming Interest Group (PPIG 2025)*. Available at `https://arxiv.org/abs/2506.23253`.

[19] R. F. Stärk (1995): *First-order theories for pure Prolog programs with negation*. *Arch. Math. Log.* 34(2), pp. 113–144, doi:10.1007/BF01270391.

[20] R. F. Stärk (1996): *Total Correctness of Logic Programs: A Formal Approach*. In R. Dyckhoff, H. Herre & P. Schroeder-Heister, editors: *ELP'96*, *LNCS* 1050, Springer, pp. 237–254, doi:10.1007/3-540-60983-0_17.

[21] R. F. Stärk (1998): *The theoretical foundations of LPTP (a logic program theorem prover)*. *Journal of Logic Programming* 36(3), pp. 241–269.

[22] Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue & Swarat Chaudhuri (2025): *CLEVER: A Curated Benchmark for Formally Verified Code Generation*. In: *2nd AI for Math Workshop @ ICML 2025*. Available at `https://openreview.net/forum?id=pqNFDA2TFm`.

[23] Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu & Zheng Wang (2025): *AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities*. arXiv:2508.11126.

[24] Song Wang (2026): *VibeContract: The Missing Quality Assurance Piece in Vibe Coding*. arXiv:2603.15691.

[25] Parker Whitfill, Cheryl Wu, Joel Becker & Nate Rush (2026): *Many SWE-bench-Passing PRs Would Not Be Merged into Main*. `https://metr.org/notes/2026-03-10-many-swe-bench-passing-prs-would-not-be-merged-into-main/`.

[26] Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta & Sanjeev Arora (2026): *AlgoVeri: An Aligned Benchmark for Verified Code Generation on Classical Algorithms*. arXiv:2602.09464.

[27] Songwen Zhao, Danqing Wang, Kexun Zhang, Jiaxuan Luo, Zhuo Li & Lei Li (2026): *Is Vibe Coding Safe? Benchmarking Vulnerability of Agent-Generated Code in Real-World Tasks*. arXiv:2512.03262.