УДК 004.052.32, 004.43 https://doi.org/10.15407/pp2026.01.004

Р.С. Шевченко, А.Ю. Дорошенко, О.А. Яценко

ANTHILL: МОВА ПРОГРЕСИВНОЇ ФОРМАЛІЗАЦІЇ ДЛЯ ПРОГРАМНИХ ПРОЄКТІВ З АГЕНТНОЮ ПІДТРИМКОЮ

У статті представлено мову Anthill — спеціалізовану мову прогресивної (поступової) формалізації знань, призначену для використання в реальних програмних проєктах з інтеграцією агентної підтримки, зокрема, великих мовних моделей. Anthill дозволяє описувати, накопичувати та підтримувати в єдиній структурованій базі знань доменні знання, функціональні та нефункціональні вимоги, проєктні рішення, інваріанти, обмеження цілісності та доведення властивостей. Відмінною особливістю є те, що база знань зберігається безпосередньо у файловій системі репозиторію проєкту, що забезпечує тісну версійовану синхронізацію специфікацій з кодом і полегшує їхню еволюцію протягом життєвого циклу розробки. Ядро мови зроблено мінімальним і складається з чотирьох базових конструктів: простір імен, сорт, правило та операція. Така архітектура аналогічна ядру сучасних систем автоматизованого доведення теорем (наприклад, Lean, Coq), де мале довірене ядро виконує лише перевірку коректності виведення, тоді як пошук доведень, генерацію правил, уточнення специфікацій та розв'язання зобов'язань делеговано зовнішнім агентам. Головною інновацією Anthill є вбудована підтримка часткової формалізації: одна й та сама мова дозволяє описувати знання на всьому спектрі — від природномовних коментарів у блоках через напівформальні машинно-перевірні правила (клаузи Хорна) до повністю формальних доведень та законів алгебри. Розглянуто три приклади застосування: структуроване управління задачами, специфікація комунікаційних протоколів для вбудованих систем та доведення властивостей на етапі компіляції.

Ключові слова: формальна специфікація, прогресивна формалізація, база знань, стигмергія, мультиагентні системи, великі мовні моделі.

R.S. Shevchenko, A.Yu. Doroshenko, O.A. Yatsenko

ANTHILL: A PROGRESSIVE FORMALIZATION LANGUAGE FOR AGENT-SUPPORTED SOFTWARE PROJECTS

The article presents the Anthill language — a specialized language for progressive (gradual) knowledge formalization, designed for use in real-world software projects with the integration of agent support, in particular large language models. Anthill allows you to describe, accumulate and maintain domain knowledge, functional and non-functional requirements, design solutions, invariants, integrity constraints and property proofs in a single structured knowledge base. A distinctive feature is that the knowledge base is stored directly in the project repository file system, which ensures close versioned synchronization of specifications with code and facilitates their evolution throughout the development life cycle. The core of the language is made minimal and consists of only four basic constructs: namespace, sort, rule and operation. This architecture is similar to the core of modern automated theorem proving systems (e. g. Lean, Coq), where a small trusted core performs only the correctness check of the derivation, while the search for proofs, rule generation, specification refinement, and resolution of commitments are delegated to external agents. The main innovation of Anthill is the built-in support for partial (progressive) formalization: one and the same language allows for the description of knowledge across the entire spectrum — from free-text natural language comments in blocks through semi-formal machine-verifying rules (Horn clauses) to fully formal proofs and laws of algebra. Three application examples are considered: structured task management, specification of communication protocols for embedded systems, and proof of properties at the compilation stage.

Keywords: formal specification, progressive formalization, knowledge base, stigmergy, multi-agent systems, large language models.

Вступ

Специфікації в сучасних програмних проєктах існують у двох крайностях. З одного боку — неформальні описи: задачі в системі управління проєктами, коментарі в коді, текстові документи з вимогами. Вони доступні людям, але не перевіряються машиною: неузгодженість між вимогою та реалізацією виявляється лише під час тестування. З іншого боку — повністю формальні специфікації в системах доведення теорем (Lean [1], Coq [2], Isabelle [3]) або алгебраїчних мовах специфікацій (OBJ [4], Maude [5]). Вони забезпечують математичну строгість, але потребують спеціалізованих знань, і поріг входу для більшості таких програмних проєктів надто високий.

Поява великих мовних моделей (Large Language Models — LLM) як учасників розробки програмного забезпечення загострює цю проблему. LLM-агенти здатні генерувати код, але працюють без стійкої структури знань: кожна сесія починається з порожнього контексту, а результати попередньої роботи зберігаються лише як текст у файлах. Відсутність формальних інваріантів означає, що агент може порушити неявні припущення проєкту, які ніде не зафіксовані машинно-перевірним чином.

Ми пропонуємо третю точку: мову *часткової формалізації з поступовим уточненням*. Anthill дозволяє починати з природномовних описів і поступово замінювати їх на формальні правила та обмеження — без необхідності формалізувати все одразу. База знань (БЗ) живе безпосередньо у файловій системі проєкту (каталог anthill/ поруч з src/ та tests/) і еволюціонує разом із кодом.

Архітектура Anthill спирається на поділ, що зарекомендував себе в системах доведення теорем: *мале довірене ядро* перевіряє правила та обмеження, а *великі недовірені агенти* (LLM, скрипти, люди) шукають рішення та докази. Ядру байдуже, як доведення було знайдене — формальним рушієм, SMT-солвером (Satisfiability Modulo Theories solver) [6], нейронною мережею чи людиною — важливо лише, що перевірка пройдена.

Стаття організована наступним чином. Розділ 1 описує принципи проєктування. У розділі 2 дається стислий огляд мови. Розділ 3 розглядає концепцію легкої аплікації в репозиторії та агентний цикл зворотного зв'язку. Розділ 4 представляє три приклади застосування, а розділ 5 обговорює зв'язок з існуючими роботами.

1. Принципи проєктування

1.1. Мінімальне ядро. Ядро мови містить рівно чотири конструкти:

- namespace — простір імен з імпортом, експортом та вкладенням;
- sort — тип (параметричний, абстрактний або визначений);
- rule — правило (клауза Хорна), єдиний примітив знань;
- operation — оголошення операції з типізованим контрактом та ефектами.

Ці конструкти дозволяють описувати багатосортні алгебри в традиції алгебраїчних мов специфікацій OBJ [4] та Maude [5]: простори імен визначають сигнатури, сорти задають носії, операції — це функціональні символи, а правила — аксіоми (тотожності та клаузи Хорна). Решта конструкцій мови зводяться до цих чотирьох. Зокрема, entity — це сорт з єдиним конструктором (алгебра з одним породжувальним елементом); fact — це правило без тіла, тобто безумовна аксіома; constraint — це правило без голови, тобто обмеження цілісності (denial у термінології логічного програмування).

Такий підхід аналогічний до архітектури систем типу proof assistant (Lean [1], Coq [2]): мале довірене ядро перевіряє виведення, тоді як тактики (великі, недовірені) шукають докази.

Ядро є мовонезалежним: сорти, правила та операції описують доменну логіку без прив'язки до конкретної мови реалізації. Ядро може бути підключене до мови-хоста через механізм підключення зовнішніх функцій (Foreign Function Interface — FFI) або реімплементоване для цієї мови; наразі підтримуються Rust та Scala. Вбудовування відбувається через механізм *реалізацій*, який відображає абстрактні

сорти на типи хост-мови, операції — на функції, а ефекти — на відповідні конструкції хост- мови.

1.2. Подання знань через правила. Всі знання в БЗ представляються у вигляді клауз Хорна:

- факт (правило без тіла): безумовна істина:

```
fact Eq{Int}
```

- правило виведення: length визначається рекурсивно:

```
rule length(nil) = 0

rule length(cons(?x, ?xs)) =
add(1, length(?xs))
```

- обмеження (правило без голови): дільник не може бути нулем:

```
constraint div_nonzero: neq(?b, 0) :-
divExact(?_, ?b)
```

Рушій виведення поєднує два механізми: SLD-резолюцію (як у Prolog) для логічного виведення над клаузами Хорна та екваціональне переписування для спрощення термів згідно з аксіомами алгебри. Хоча це різні обчислювальні моделі (резолюція працює з цілями та підстановками, а переписування — з напрямленими тотожностями), на рівні реалізації обидва зводяться до єдиної базової операції: співставлення терму з патерном та застосування підстановки. Цей підхід відтворює архітектуру системи TermWare [7], де правила переписування та логічне виведення об'єднані через контекстні терми з ефективною диспетчеризацією.

Кожний факт несе метадані: хто його створив, коли, з яким рівнем довіри та в якій ітерації. Кожний факт несе рівень довіри, що може поступово підвищуватися: від початкового припущення через підтвердження тестами, верифікацію автоматичним інструментом (наприклад, SMT-солвером) до повного формального доведення, перевіреного довіреним ядром.

1.3. Часткова формалізація. Будь-яке оголошення може мати один або декілька блоків опису ({< >}) — вільнотекстові описи, що зберігаються як факти в БЗ. На відміну від коментарів, блоки опису є структурними: вони доступні для запитів та аналізу. В поєднанні з анонімними змінними (?) це дає спектр від повністю неформального до повністю формального:

- повністю неформальне:

```
sort Account
  {< Банківський рахунок.
     Має баланс, власника,
     та підтримує операції зняття та
     поповнення. >}
  sort T = ?
end
```

- частково формальне:

```
sort Account
  {< Банківський рахунок >}

  entity account(id: AccountId,
  owner: String, balance: Money)

  operation withdraw(amount: Money)
  -> Account

  constraint positive_balance:
  gte(?bal, 0) :- account(?_, ?_, ?bal)
end
```

Всюди, де очікується ім'я типу або вираз, можна використовувати логічні змінні: ?x — іменована змінна (спільна в межах області видимості); ? — анонімна (кожне входження позначає окрему змінну). Зокрема, sort T = ? визначає абстрактний параметр типу — сорт, що буде уточнений під час інстанціації.

2. Огляд мови

Цей розділ дає стислий огляд мови, достатній для розуміння прикладів. Повна специфікація доступна як частина репозиторію проєкту [8].

2.1. Простори імен та модульність. Простори імен організують код ієрархічно, з імпортом та експортом символів:

```
namespace anthill.prelude.Int
  import anthill.prelude.{Eq, Ordered,
                          Numeric}
  export abs, neg, mod, rem, sign,
         divExact
  fact Eq{Int}
```

```
  fact Ordered{Int}
  fact Numeric{Int}
end
```

2.2. Сорти. Сорти описують типи. Параметричний сорт має абстрактні параметри (sort T = ?):

```
sort anthill.prelude.List

  export List, nil, cons, length,
         member, append

  sort T = ?  -- параметр типу
              -- (абстрактний)

  entity nil   -- порожній список

  -- клітинка
   entity cons(head: T, tail: List)

  operation length(l: List) -> Int

  rule length(nil) = 0

  rule length(cons(?x, ?xs)) =
  add(1, length(?xs))

end
```

Задоволення специфікації декларується через факти. Запис fact Eq{Int} означає, що тип Int задовольняє специфікацію Eq — операції eq та neq доступні для значень Int. На відміну від ключового слова instance у мові Haskell, тут задоволення специфікації — це звичайний факт в БЗ, який може мати метадані та рівень довіри.

2.3. Специфікації та вимоги. Специфікації — це сорти з абстрактними операціями та правилами-законами:

```
sort anthill.prelude.Ordered
  sort T = ?
  requires Eq{T}
  operation compare(a: T, b: T) -> Int

  -- Похідні операції
  operation gt(a: T, b: T) -> Bool
  rule gt(?a, ?b) =
  gt(compare(?a, ?b), 0)

  -- Закони
  rule compare_antisym:
  compare(?a, ?b) =
  neg(compare(?b, ?a))
  rule compare_refl:
  compare(?a, ?a) = 0

end
```

Зв'язок requires Eq{T} означає: для будь-якого типу, що реалізує Ordered, має існувати факт Eq{T}.

2.4. Операції та ефекти. Операції мають типізований контракт та оголошення ефектів:

```
operation withdraw(account: Account,
          amount: Money) -> Account
  requires gte(amount, 0)
  ensures gte(balance(result), 0)
  effects (Modify(ledger))
```

Ефекти явно декларують, які ресурси операція модифікує, читає або може перервати. Це забезпечує можливість статичного аналізу залежностей між - операціями.

3. Аплікація в репозиторії та агентний цикл

Anthill-застосунок — це набір .anthill файлів, що живуть безпосередньо в репозиторії проєкту поруч з src/ та tests/, еволюціонують разом з кодом і версіонуються в тому ж репозиторії.

Центральне поняття агентної взаємодії — *зобов'язання* (obligation): логічне твердження, яке має бути виконане. Це може бути твердження, яке потрібно довести; операція без реалізації; обмеження, для якого потрібно побудувати свідчення; або декомпозиція складної задачі на підзадачі. Зобов'язання формулюються як звичайні терми та правила мови Anthill. Агент (LLM, скрипт, людина або формальний рушій) бере зобов'язання та надає відповідь — доведення, реалізацію, факт з відповідним рівнем довіри. Ядро перевіряє, чи відповідь задовольняє зобов'язання.

Координація між агентами відбувається через спільне середовище — базу знань — без центрального оркестратора (принцип *стигмергії*). Стан БЗ визначає поточну роботу: агент спостерігає відкриті зобов'язання, порушені обмеження, операції без реалізації — і обирає, що виконувати. Результат записується як факт з метаданими походження (провенансу). Якщо факт визнано некоректним, виникає *кон-*

тамінація: залежні факти автоматично втрачають довіру, блокуючи подальшу роботу, поки неузгодженість не буде усунена.

Формалізація не відбувається одномоментно — різні частини проєкту можуть перебувати на різних рівнях: від вільнотекстових описів (рівень 0), через доменну модель як багатосортну алгебру (рівень 1), формальні обмеження та специфікації з законами (рівень 2), до формальних доведень з верифікацією зовнішнім рушієм (рівень 3). Зобов'язання та агентний цикл працюють однаково на всіх рівнях.

4. Приклади застосування

4.1. Декомпозиція задач як зобов'язання. Зобов'язання в Anthill можуть описувати декомпозицію задач — розбиття складної цілі на підзадачі з формальними залежностями та критеріями прийому. Кожна задача — це факт в БЗ:

```
workitem WI-AUTH-001 {
  description: "Define User entity and
                authentication traits"
  acceptance: Compiles({ path: "src",
                         scope: Main })
  depends_on: []
  status: Open
  [trust: proposed, agent: "architect"]
}

workitem WI-AUTH-002 {
  description: "Implement JWT token
                generation and validation"
  acceptance: Compiles({path:"src"}),
  ToolPasses(cargo-test)
  depends_on: [WI-AUTH-001]
  status: Open
  [trust: proposed, agent: "architect"]
}
```

Правила виведення визначають логіку робочого процесу — наприклад, які задачі готові до виконання:

```
rule claimable(?id, ?desc)
  :- WorkItem(id: ?id, status: Open,
              description: ?desc),
     all_deps_verified(?id)

rule blocked(?id, ?desc)
  :- WorkItem(id: ?id, status: Open,
              description: ?desc),
     not(all_deps_verified(?id))
```

Запит claimable(?id, ?desc) — це логічний висновок над станом БЗ через SLD-резолюцію. Та ж механіка працює і для більш формальних зобов'язань: «які обмеження порушені», «які операції не мають реалізації».

4.2. Специфікація комунікаційних протоколів. Розглянемо більш формальний приклад: специфікація правил та обмежень для системи моделювання безпілотних апаратів. У роботі [9] симуляційне середовище (на базі Bevy ECS) взаємодіє з агентами (Python, Scala) через gRPC, передаючи повідомлення за протоколом MAVLink. Поведінка агента — це функція, що на основі поточних показів сенсорів та історії видає команди актуаторам. Агенти реалізуються різними мовами, але мають відповідати спільним обмеженням безпеки та протоколу.

Anthill дозволяє формалізувати три категорії знань, які в поточній реалізації існують лише неявно, а саме: обмеження місії, автомат станів MAVLink і правила деградації сенсорів.

Обмеження місії. Місія (пошук, патрулювання, доставка) має властивості, які мають виконуватися протягом усього польоту:

```
namespace blefusku.mission
  import anthill.prelude.{Int, Float,
                          Bool, List}

  entity DroneState(id: String, position:
Vector3D, altitude: Float, battery: Float)

  -- Дрон має залишатися в межах
  -- геозони
  constraint geofence:
    in_zone(?pos, ?zone)

    :- DroneState(id: ?d, position: ?pos),
       Mission(drone: ?d, zone: ?zone)
  -- Безпечна відстань між дронами
```

```
  constraint safe_separation:
    distance(?p1, ?p2) > 100.0
    :- DroneState(id: ?d1, position:
                  ?p1),
       DroneState(id: ?d2, position: ?p2),
       ?d1 != ?d2

  -- Заряд батареї має бути достатнім
  -- для повернення

  constraint battery_reserve:
    ?battery > return_energy(?pos,
    ?home)
    :- DroneState(id: ?d, position: ?pos,
                  battery: ?battery),
       HomeBase(drone: ?d, position:
                  ?home)
end
```

Автомат станів MAVLink. Протокол MAVLink визначає автомат станів автопілота (Disarmed → Armed → Takeoff → Guided → Land), де певні команди допустимі лише в певних станах. Ці правила наразі розкидані по документації та реалізаціях:

```
namespace blefusku.mavlink_fsm

  -- Допустимі переходи між
  -- станами

  rule valid_transition(Disarmed,
     Armed) :- PrearmChecks(ok)

  rule valid_transition(Armed,
                        Takeoff)

  rule valid_transition(Takeoff,
                        Guided)
    :- DroneState(altitude: ?a), ?a >
       min_takeoff_alt

  rule valid_transition(Guided, Land)

  rule valid_transition(Land,
                        Disarmed)

    :- DroneState(altitude: ?a), ?a < 0.5

  -- Команда допустима лише у
  -- відповідному стані

  constraint valid_command:
    valid_in_state(?cmd, ?state)
      :- CommandSent(command: ?cmd),
         FlightState(state: ?state)
end
```

Правила деградації сенсорів. Поведінка у разі втраті GPS-сигналу або дрейфі IMU — це доменне знання, яке має бути явним:

```
namespace blefusku.sensors
  rule gps_reliable(?d)
    :- SensorData(drone: ?d, type: gps,
                  satellites: ?n, fix_type: ?f),
       ?n >= 6, ?f >= 3

  rule navigation_mode(?d,
                       gps_primary)
    :- gps_reliable(?d)

  rule navigation_mode(?d,
                  imu_dead_reckoning)
    :- not(gps_reliable(?d)),
       SensorData(drone: ?d, type: imu)

  -- в режимі Guided дрон має мати
  -- навігацію
  constraint must_have_navigation:
    navigation_mode(?d, ?)
    :- DroneState(id: ?d, state: Guided)
end
```

Сформульовані правила та обмеження можуть використовуватися двома способами. По-перше, *верифікація трас*: записані траси виконання агентів (послідовності станів, сенсорних даних та команд) завантажуються в БЗ як факти, після чого перевіряються обмеження — порушення виявляються перед- і пост- умови автоматично. По-друге, *аналіз коду агентів*: обмеження Anthill можуть бути відображені на перед- і пост- умови хост- мови (наприклад, requires/ensures в Scala або assert в Python), що дозволяє перевіряти їх статично або під час тестування.

Зв’язок з інтерфейсно-орієнтованим підходом до моделювання мультиагентних систем [9]: Anthill надає формальний шар специфікації поверх gRPC-інтерфейсів [10], дозволяючи виражати інваріанти, які не можуть бути описані лише засобами protobuf [11], — обмеження

місії, автомат станів протоколу та правила деградації.

4.3. Доведення на етапі компіляції. Попередні приклади ілюструють використання Anthill для специфікації доменних знань. В рамках поточної роботи ми також досліджуємо використання БЗ Anthill як рушія доведень, вбудованого безпосередньо в компіляційний конвеєр мови-хоста.

Ідея полягає у створенні системи верифікації властивостей на етапі компіляції для Scala 3. Центральним є тип Checked[A, P], що позначає значення типу A, для якого доведено предикат P. Предикати записуються як типорівневий AST: @pre(_ > 0) на параметрі означає, що значення має тип Checked[A, GT[Self, Lit[0]]], а @post(_ >= 0) на результаті — що повернене значення несе доведення невід'ємності.

Арифметичні операції над Checked-значеннями оголошують *зобов'язання доведення* як using-параметри — наприклад, def +[Q](b: Checked[A, Q])(using proof: ProofAdd[P, Q]): Checked[A, proof.Out]. Компілятор шукає відповідний given-instance через механізм implicit resolution, який є рушієм зворотного виведення (backward chaining): given відповідає правилу виведення, fact — аксіомі, summon — зобов'язанню доведення.

Прості доведення вирішуються прямими правилами given-instances (наприклад, given mulNonNeg: ProofMul[NonNeg, NonNeg] with { type Out = NonNeg }). Коли прямих правил недостатньо, компілятор звертається до macro-given, який делегує до БЗ Anthill.

Предикати типорівневого AST в Scala відображаються на терми Anthill (наприклад, GT[Self, Lit[0]] стає gt(self, lit(0))), а правила доведення зберігаються в БЗ. Наведемо приклад: набір правил, що описує безпечний доступ до масиву, де потрібно довести, що індекс знаходиться в межах.

```
namespace ctproof.array_safety
  import anthill.prelude.{Int, Bool}
  -- Базові аксіоми послаблення та
транзитивності
  rule ?a >= ?b :- ?a > ?b
  rule ?a >= ?c :- ?a >= ?b, ?b >= ?c

  -- Індекс в межах масиву
  rule in_bounds(?i, ?n) :- ?i >= 0,
                            ?i < ?n
  -- Якщо індекс в межах
  -- i n == length(arr),

  -- то доступ arr(i) безпечний
  rule safe_access(?arr, ?i)
    :- in_bounds(?i, ?n), ?n =
       length(?arr)

  -- Індукційний крок: доступ до
  -- хвоста потребує нового
  -- доведення зі зменшеним
  -- індексом

  Rule requires_proof(
    safe_access(?arr2, ?i - 1))

    :- safe_access(?arr, ?i),
      ?arr2 = tail(?arr)

  -- Ціле ділення: якщо дільник
  -- додатний і ділене невід'ємне,
  -- результат невід'ємний

  rule (?a / ?b) >= 0 :- ?a >= 0, ?b > 0

  -- Композиція: якщо f зберігає P,
  -- і g зберігає Q, то compose(g, f)
  -- зберігає and(P, Q)

  rule preserves(compose(?g, ?f), ?p
                          and ?q)
    :- preserves(?f, ?p), preserves(?g,
                                    ?q)
end
```

Коли Scala-компілятор зустрічає операцію над Checked-значенням (наприклад, arr(i) де arr: Checked[Array[T], _]), макрос генерує зобов'язання доведення safe_access(arr, i). Спочатку доведення шукається в контентно-адресованому кеші: якщо для даного зобов'язання вже існує збережене доведення і його передумови не змінилися, воно повторно перевіряється ядром і використовується без повторного пошуку. Якщо кешоване доведення відсу-

тнє або стало невалідним (наприклад, через зміни в коді, що вплинули на передумови), зобов'язання передається до БЗ Anthill для пошуку нового доведення. Якщо і було отримано з конструкції if (i < arr.length), передумова (i >= 0) and (i < length(arr)) вже є в контексті, і доведення будується автоматично через SLD-резолюцію. Для арифметичних підцілей, що виходять за межі можливостей резолюції, викликається Z3 SMT-солвер як вбудована операція.

Таким чином, Anthill працює як *вбудований рушій доведень* — активна частина компіляційного конвеєра, а не лише формат зберігання знань. Трирівнева архітектура (пряма резолюція given → SLD-резолюція Anthill → Z3 SMT) дозволяє балансувати між швидкістю та потужністю: більшість доведень вирішуються на першому рівні без накладних витрат.

5. Огляд суміжних досліджень

Anthill знаходиться в традиції алгебраїчних мов специфікацій OBJ [4], CafeOBJ [12], Maude [5] та використовує архітектурний принцип proof assistants Lean 4 [1], Coq [2], Isabelle [3] — мале довірене ядро перевіряє докази, тоді як великі недовірені тактики їх шукають.

Питання формальних гарантій для ШІ-агентів набуває актуальності в кількох напрямках. Universalis [13] — це мова для програмного синтезу класу AI-first, побудована на Prolog, де перед- та пост-умови вбудовані в семантику як механізм безпеки ШІ. В роботі [14] пропонують capture checking в Scala 3 як оболонку "safety harness" для агентів — статичне відстеження спроможностей (capabilities) запобігає витоку інформації та несанкціонованим побічним ефектам. Обидва підходи підтверджують тенденцію до використання формальних методів замість суто статистичних методів вирівнювання (RLHF).

В роботі [15] надається огляд взаємодії формальних методів та LLM: автоформалізація, LLM-керований пошук доведень, нейронне доведення теорем (AlphaProof [16]). В екосистемі Lean 4 це реалізовано в проєктах Lean Copilot [17] та APOLLO [18] — агентних фреймворках, де LLM інтегровані як тактики пошуку доведень. Окремий напрямок — використання машинного навчання як евристики для прискорення формального пошуку: графові нейронні мережі (Graph neural networks — GNN) для ранжування рівнянь у солвері [19], scored logic monad для направлення логічного бектрекінгу [20].

Висновки

Представлено мову Anthill для прогресивної формалізації знань у програмних проєктах. Мова дозволяє описувати багатосортні алгебри з аксіомами у вигляді клауз Хорна. Ключовою є підтримка часткової формалізації: спектр від вільнотекстових описів до формальних доведень в одній мові з єдиним механізмом рівнів довіри та провенансу.

Наведені приклади демонструють застосування на різних рівнях формалізації: від декомпозиції задач із машинно-перевірними критеріями, через специфікацію обмежень протоколів та правил деградації для мультиагентних систем, до верифікації властивостей на етапі компіляції з використанням БЗ як рушія доведень.

Повна специфікація мови та реалізація доступні за адресою https://github.com/rssh/anthill.

Література

1. L. de Moura, S. Ullrich, The Lean 4 theorem prover and programming language, *28th International Conference on Automated Deduction (CADE 2021). Lecture Notes in Computer Science.* 2021. Vol. 12699. P. 625–635. https://doi.org/10.1007/978-3-030-79876-5_37
2. The Coq Development Team. The Coq Proof Assistant. Accessed: 18.03.2026. https://coq.inria.fr
3. T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL — a proof assistant for higher-order logic, Springer, 2002.
4. J. Goguen, T. Winkler. Introducing OBJ. Technical Report, SRI International, 1988.
5. M. Clavel et al., Maude: specification and programming in rewriting logic, *Theoretical Computer Science*. 2002. Vol. 285, N 2. P. 187–243. https://doi.org/10.1016/S0304-3975(01)00359-0

6. L. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science. 2008. Vol. 4963. P. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
7. Р. С. Шевченко, А. Ю. Дорошенко, TermWare-3 — система переписування термів, заснована на контекстному численні, *Проблеми програмування*. 2019. № 1. https://doi.org/10.15407/pp2019.01.048
8. Anthill kernel language specification. Accessed: 18.03.2026. https://github.com/rssh/anthill
9. Р. С. Шевченко, А. Ю. Дорошенко, В. О. Лесик, О. В. Савчук, О. А. Яценко, Інтерфейсно-орієнтований підхід до засобів моделювання мультиагентних систем, *Проблеми програмування*. 2025. № 1. С. 110–117. https://doi.org/10.15407/pp2025.01.110
10. gRPC. Accessed: 18.03.2026. URL: https://grpc.io/
11. Protocol Buffers. Accessed: 18.03.2026. https://protobuf.dev
12. R. Diaconescu, K. Futatsugi, CafeOBJ Report. World Scientific, 1998.
13. E. Meijer, Unleashing the power of end-user programmable AI, *ACM Queue*. 2025. Vol. 23, N 3. Accessed: 18.03.2026. https://spawn-queue.acm.org/doi/10.1145/3746223
14. M. Odersky, Y. Zhao, Y. Xu, O. Bračevac, C.N. Pham, tracking capabilities for safer agents, *arXiv:2603.00991*. 2026. P. 1–21. https://doi.org/10.48550/arXiv.2603.00991
15. K. Yang, G. Poesia, J. He, W. Li, K. Lauter, S. Chaudhuri, D. Song, Formal mathematical reasoning: a new frontier in AI, *arXiv:2412.16075*. 2025. P. 1–42. https://doi.org/10.48550/arXiv.2412.16075
16. Google DeepMind. AI achieves silver-medal standard solving International Mathematical Olympiad problems (AlphaProof announcement). 2024. Accessed: 18.03.2026. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/
17. P. Song et al., Lean Copilot: large language models as copilots for theorem proving in Lean, *arXiv:2404.12534*, 2024. P. 1–26. https://doi.org/10.48550/arXiv.2404.12534
18. X. Hu et al., APOLLO: automated LLM and Lean collaboration for advanced formal reasoning, *arXiv:2505.05758*. 2025. P. 1–28. https://doi.org/10.48550/arXiv.2505.05758
19. P. A. Abdulla, M. F. Atig, J. Cailler, C. Liang, P. Rümmer, When GNNs met a Word equations solver: learning to rank equations, *arXiv:2506.23784*, 2025. https://doi.org/10.48550/arXiv.2506.23784
20. R. Shevchenko, A. Doroshenko, O. Yatsenko, A. Nemish, Merging logic and the coinductive selection monad: mixing machine learning into logical search. *ICTERI 2025. Communications in Computer and Information Science*. Vol. 2763. 2025. P. 33–46. https://doi.org/10.1007/978-3-032-10477-9_3

References

1. L. de Moura, S. Ullrich, The Lean 4 theorem prover and programming language, in: *28th International Conference on Automated Deduction (CADE 2021), Lecture Notes in Computer Science* 12699 (2021) 625–635. doi: 10.1007/978-3-030-79876-5_37
2. The Coq Development Team. The Coq Proof Assistant. Accessed: 18.03.2026. https://coq.inria.fr
3. T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL — a proof assistant for higher-order logic, Springer, 2002.
4. J. Goguen, T. Winkler. Introducing OBJ. Technical Report, SRI International, 1988.
5. M. Clavel et al., Maude: specification and programming in rewriting logic, in: *Theoretical Computer Science* 285 (2002) 187–243. doi: 10.1016/S0304-3975(01)00359-0
6. L. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science 4963 (2008) 337–340. doi: 10.1007/978-3-540-78800-3_24
7. R. S. Shevchenko, A.Yu. Doroshenko, TermWare-3 – term rewriting system, based on context-term calculus, *Problems in programming* 1 (2019) 48–58. https://doi.org/10.15407/ pp2019.01.048 [in Ukrainian]
8. Anthill kernel language specification. Accessed: 18.03.2026. https://github.com/rssh/anthill
9. R.S. Shevchenko, A.Yu. Doroshenko, V.O. Lesyk, O.V. Savchuk, O.A. Yatsenko, Interface-oriented approach to modelling tools for multi-agent systems, *Problems in programming* 1 (2025) 110–117.

https://doi.org/10.15407/pp2025.01.110 [in Ukrainian]
10. gRPC. Accessed: 18.03.2026. URL: https://grpc.io/
11. Protocol Buffers. Accessed: 18.03.2026. https://protobuf.dev
12. R. Diaconescu, K. Futatsugi, CafeOBJ Report. World Scientific, 1998.
13. E. Meijer, Unleashing the power of end-user programmable AI, in: *ACM Queue* 23 (2025) Accessed: 18.03.2026. https://spawn-queue.acm.org/doi/10.1145/3746223
14. M. Odersky, Y. Zhao, Y. Xu, O. Bračevac, C.N. Pham, tracking capabilities for safer agents, in: *arXiv:2603.00991* (2026) 1–21. doi: 10.48550/arXiv.2603.00991
15. K. Yang, G. Poesia, J. He, W. Li, K. Lauter, S. Chaudhuri, D. Song, Formal mathematical reasoning: a new frontier in AI, in: *arXiv:2412.16075* (2025) 1–42. doi: 10.48550/arXiv.2412.16075
16. Google DeepMind. AI achieves silver-medal standard solving International Mathematical Olympiad problems (AlphaProof announcement). 2024. Accessed: 18.03.2026. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/
17. P. Song et al., Lean Copilot: large language models as copilots for theorem proving in Lean, *arXiv:2404.12534* (2024) 1–26. doi: 10.48550/arXiv.2404.12534
18. X. Hu et al., APOLLO: automated LLM and Lean collaboration for advanced formal reasoning, *arXiv:2505.05758* (2025) 1–28. doi: 10.48550/arXiv.2505.05758
19. P. A. Abdulla, M. F. Atig, J. Cailler, C. Liang, P. Rümmer, When GNNs met a Word equations solver: learning to rank equations, in: *arXiv:2506.23784* (2025) doi: 10.48550/arXiv.2506.23784
20. R. Shevchenko, A. Doroshenko, O. Yatsenko, A. Nemish, Merging logic and the coinductive selection monad: mixing machine learning into logical search, in: *ICTERI 2025. Communications in Computer and Information Science* 2763 (2025) 33–46. doi: 10.1007/978-3-032-10477-9_3

Дата першого надходження до видання: 12.03.2026
Внутрішня рецензія отримана: 17.03.2026
Зовнішня рецензія отримана: 18.03.2026
Дата прийняття статті до друку: 19.03.2026
Дата публікації: 16.04.2026

Про авторів:

[1]Шевченко Руслан Сергійович,
кандидат технічних наук
старший науковий співробітник
Shevchenko Ruslan,
Ph.D (technical sciences), senior scientist
https://orcid.org/0000-0002-1554-2019.

[1,2]Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор,
провідний науковий співробітник
Doroshenko Anatoliy,
Ph.D (doctor, physical and mathematical sciences), professor, leading scientist
http://orcid.org/0000-0002-8435-1451.

[1]Яценко Олена Анатоліївна,
кандидат фізико-математичних наук,
старший науковий співробітник
Yatsenko Olena,
Ph.D (physical and mathematical sciences), senior scientist
http://orcid.org/0000-0002-4700-6704.

Місце роботи авторів:

[1] Інститут програмних систем
НАН України,
[1] Institute of Software Systems.
National Academy of Sciences of Ukraine
тел. +38-067-407-32-33
E-mail: doroshenkoanatoliy2@gmail.com,
ruslan@shevchenko.kiev.ua,
oayat@ukr.net
Сайт: https://iss.nas.gov.ua

[2] Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»
[2] National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”
Сайт: https://ist.kpi.ua