Теоретичні основи побудови формальних моделей програм та методів верифікації програм

Третяков Єгор

2025

Зміст

1 Основні поняття програмування

1.1 Властивості основної пентади програмування

Пентада основних понять програмування об'єднує п'ять фундаментальних понять: користувача, проблему, програму, процес виконання та процес програмування. Згідно з Нікітченком, ця пентада виділяє ключові аспекти кожного поняття. Зокрема, для поняття *програми* основними зовнішніми аспектами є:

- **Адекватність** — здатність програми правильно відображати зміст початкової проблеми;
- **Прагматичність** — ефективність і практична доцільність рішення, заданого програмою;
- **Обчислюваність** — можливість алгоритму програми бути виконаним машинним шляхом (дискретною обчислювальною моделлю);
- **Генетичність** — властивість програми мати історію походження (метадані) або можливість відтворення через системи версій.

Для поняття *процесу програмування* Нікітченко виділяє такі основні аспекти:

- **Проблемна орієнтованість** — спрямованість процесу на вирішення конкретної задачі;
- **Експлікативність** — чіткість та прозорість опису програмного рішення (зрозумілість коду та пояснень).

Тобто основна пентада програмування акцентує увагу на зовнішніх характеристиках поняття *програма* (адекватність, прагматичність, обчислюваність, генетичність) та поняття *процес програмування* (проблемна орієнтованість, експлікативність):contentReference[oaici
Ці властивості показують, яким чином програма вирішує задачу, наскільки результат вважається коректним і ефективним, а також на які моменти слід зважати в процесі розроблення програмного забезпечення.

1.2 Властивості програмної пентади

Програмна пентада описує п'ять базових елементів програмного середовища: *дане* (значення вхідних параметрів), *функція* (обчислювана функція), *ім'я функції* (ідентифікатор функції), *композиція* (поєднання функцій) та *дескрипція* (опис функції через ім'я). За Нікітченком, ці п'ять понять характеризують основні властивості програм на абстрактному рівні та утворюють пентаду основних програмних понять:contentReference[oaicite:2]index=2. У рамках цієї пентади виокремлюють три взаємопов'язані аспекти програми:

- **Семантичний аспект** — зв'язок *даних*, *функцій* і *композицій*; цей аспект описує значення (семантику) програми з точки зору обчислюваних функцій;
- **Синтаксичний аспект** — зв'язок *даних*, *імен* та *дескрипцій*; він стосується формального опису та структури програми (синтаксису);

- **Денотаційний аспект** — зв'язок *імен* і *дескрипцій* з *функціями* та *композиціями*; цей аспект поєднує імена та описи з обчислюваними функціями, забезпечуючи відповідність між назвою програми та її змістом:contentReference[oaicite:3]index=3.

Таким чином, програмна пентада демонструє, як структурні елементи програми (дані, імена, обчислювані функції) поєднуються в різних «програмахних» аспектах (семантичному, синтаксичному, денотаційному). Відзначимо також, що ця пентада формувалася в два кроки: спочатку тріада «користувач–проблема–програма», а потім розгортання функції та імені в поняття композиції та дескрипції:contentReference[oaicite:4]index=4.

1.3 Часткова коректність програм

Часткова коректність програми означає: якщо програма стартує з певного початкового стану та завершується, то її результат відповідає специфікованій меті. Формально, нехай P — властивість (передумова) початкового стану, Q — властивість (постумова) кінцевого стану, і S — програма. Тоді часткова коректність означає, що для всіх вхідних станів st з класу ST виконується

$$\forall st \in ST \ \big(\texttt{termination}(S, st) \ \Rightarrow \ \texttt{correctness}(S, st)\big),$$

де $\texttt{termination}(S, st)$ означає, що виконання S завершується на стані st, а $\texttt{correctness}(S, st)$ означає, що після завершення програми S з початкового стану st виконується властивість Q:contentReference[oaicite:5]index=5. На практиці доведення часткової коректності здійснюють за допомогою аксіоматичного підходу Флойда–Хоара. Для кожного конструктивного елемента програми формулюють логічні правила (аксіоми):

- *Присвоєння:* правило таке, що для команди `x := E` справедлива трійка $\{Q[x/E]\}$ `x := E` $\{Q\}$, де вираз $Q[x/E]$ отримано підстановкою E замість x у Q;
- *Пропуск (skip):* $\{P\}$ `skip` $\{P\}$ для будь-якої формули P;
- *Послідовність:* з $\{P\}\ S_1\ \{R\}$ і $\{R\}\ S_2\ \{Q\}$ випливає $\{P\}\ S_1; S_2\ \{Q\}$;
- *Умова:* якщо $\{P \wedge B\}\ S_t\ \{Q\}$ і $\{P \wedge \neg B\}\ S_f\ \{Q\}$, то $\{P\}$ `if` B `then` S_t `else` $S_f\ \{Q\}$;
- *Цикл:* для циклу `while` B `do` S потрібно знайти інваріант I, такий що $\{I \wedge B\}\ S\ \{I\}$, тоді $\{I\}$ `while` B `do` $S\ \{I \wedge \neg B\}$.

Доведення проводять індуктивно по структурі програми, підбираючи інваріанти для циклів. Якщо всі трійки за цими правилами виведені, то доводиться, що з передумови P та умов початку програми випливає постумова Q за умови завершення програми.

1.4 Повна коректність програм

Повна коректність (тотальна коректність) означає, що програма не лише при припущенні її завершуваності дає правильний результат, але й сама завжди завершується на всіх зазначених вхідних даних. Формально повна коректність описується умовою

$$\forall st \in ST \ \big(\texttt{termination}(P, st) \wedge \texttt{correctness}(P, st)\big),$$

що означає і виконання специфікації (`correctness`) і гарантію завершення (`termination`) для всіх $st \in ST$:contentReference[oaicite:6]index=6. Щоб довести повну коректність,

спочатку доводять часткову коректність (як описано вище), а потім окремо обґрунтовують відсутність нескінченних зациклень. Для цього часто вводять функцію рангу (варіанту циклу) — добре упорядковану міру, яка на кожному кроці циклу зменшується і не може спадати довше за певний час. Наприклад, для циклу з лічильником i часто використовують, що i зменшується і обмежений знизу нулем. Доказ завершення може базуватися на леміях про допустимість рекурсивного опису або на фундаментальній конструкції индукції по природних числах.

1.5 Формалізація поняття програми

Формалізація поняття програми полягає в описі програми у строгих математичних термінах, виділенні її синтаксичних і семантичних атрибутів. Згідно з Нікітченком, ключовим є *принцип теоретико-функціональної формалізації*: програми розглядаються як функції, що діють на відповідних множинах даних на різних рівнях абстракції:contentReference[oaicite:7]index=7. Так, програма може бути формалізована як функція (часткова чи тотальна) зі станів у стани, тобто її семантика задається у вигляді відображення в просторі станів. Крім того, вводять *класи функцій* (наприклад, номінативні, булеві, функції перетворення станів), щоб охарактеризувати типи обчислювань. Також використовують поняття *програмної системи* — набір конструктивних засобів (операцій і функцій), що задають семантику мови програмування. Наприклад, номінативний рівень формалізації передбачає опис програм як композиційно-дескриптивних систем, що дозволяє описати семантику мов програмування та прикладних логічних мов:contentReference[oaicite:8]index=8. Таким чином, формалізація програми включає побудову формальної моделі (синтаксичної алгебри, семантичної області) і суворого визначення обчислюваних функцій та властивостей, які вона має задовольняти.

1.6 Класи функцій

Узагальнено виділяють кілька класів функцій, важливих для опису поведінки програмних систем. Зокрема, за Нікітченком вводяться класи функцій програмної алгебри A_{Prog}:contentReference[oaicite:9]index=9:

- F_{NA} — клас n-арних номінативних арифметичних функцій (збереження числових чи інших базових значень);
- F_{NB} — клас булевих (предикатних) функцій на номінативних даних;
- F_{NAB} — клас біномінативних функцій (поєднань арифметичних і булевих операцій);
- F_A — клас номінативних функцій значень (з типом A — абстрактні значення);
- F_B — клас номінативних булевих функцій (значень типу `boolean`);
- F_S — клас функцій перетворення станів (відображень $State \to State$).

Ці класи об'єднані в *алгебру функцій* $A_{\text{Prog}} = \langle F_{NA}, F_{NB}, F_{NAB}, F_A, F_B, F_S;\ \text{операції} \rangle$ (на рис. 1.4):contentReference[oaicite:10]index=10. До операцій цієї алгебри належать суперпозиція функцій, присвоєння (оновлення стану), послідовне виконання, умовна

функція та цикл. Таким чином, різні класи функцій відображають способи обчислень: F_A, F_B задають базові обчислювальні одиниці на даних, а F_S — перетворення глобального стану. Комозиції цих функцій і операцій утворюють замкнену систему обчислюваних функцій, яка є основою формальної семантики програм.

1.7 Програмні системи різного рівня абстракції

Нікітченко виділяє три основні рівні абстракції програмних систем:contentReference[oaicite:11]index

- **Абстрактний рівень.** Данні та функції тут розглядаються у найзагальнішому вигляді, без конкретного відображення. Програмна система абстрактного рівня описується як $ADS = \langle D \rangle$, де D — довільна множина даних. В операціях використовуються лише істинні/хибні умови та булеві оператори у базовому розумінні.

- **Булевий рівень.** Введено спеціальні константи та операції над булевими значеннями. Система даних булевого рівня (BDS) позначається $\langle D, \texttt{true}, \texttt{false}, c_1, \ldots, c_k \rangle$, де додають базові константи. За Нікітченком, т.зв. *перед-булева*, *булева* та *пост-булева* системи визначаються різними наборами констант (наприклад, тільки `true`, або обидві `true`, `false`, тощо):contentReference[oaicite:12]index=12.

- **Номінативний рівень.** Данні представляються як номінативні дані (значення з іменованих відображень). Тут ND — клас ієрархічних номінативних даних (див. нижче). Програмна система номінативного рівня описується як композиційно-дескриптивна: використовуються операції складання та зв'язування даних за іменами. Саме на цьому рівні побудовано основний формалізм опису програм, адже він дозволяє відобразити реальні структури даних (масиви, записи, множини тощо) і операції над ними:contentReference[oaicite:13]index=13.

Тож програмні системи різних рівнів відповідають різній грануляції опису: від узагальненого ідеалізованого (абстрактного) до конкретно назвного (номінативного) представлення даних. Номінативний рівень, особливо в його композиційно-номінативній формі, є основою для реального опису мов програмування та їх семантики:contentReference[oaicite:1

1.8 Клас номінативних даних

Номінативні дані — це ієрархічні структури, які задаються за допомогою множини імен V та базових значень W. Згідно з визначенням, клас номінативних даних ND будується індуктивно:contentReference[oaicite:15]index=15:

$$ND(0) = W, \qquad ND(i+1) = V \to ND(i), \quad i \in \omega,$$

і, отже,

$$ND = \bigcup_{i \in \omega} ND(i).$$

Інакше кажучи, $ND = W \cup (V \to ND)$:contentReference[oaicite:16]index=16. Це означає, що номінативні дані складаються з елементів W (атоми) та скінченних відображень із множини імен V у попередньо визначений клас ND. У практичному програмуванні розрізняють:

- *Іменні дані* (однозначні номінативні дані) — скінченні відображення з унікальними іменами (наприклад, записи, масиви, таблиці, файли);
- *Номінативні дані* у вузькому сенсі — скінченні відображення з можливими дублями імен (наприклад, множини, передмножини, мультимножини тощо):contentReference[oai

Таким чином, номінативні дані формалізують іменовані зв'язки між значеннями: кожний рівень побудови є відображенням, де область визначення — імена, а значення — або атомарні елементи W, або інші номінативні дані. Наприклад, множину $\{a, b, c\}$ можна подати як іменоване відображення $[1 \to a,\ 1 \to b,\ 1 \to c]$, тобто всі елементи мають стандартне ім'я 1:contentReference[oaicite:18]index=18.

1.9 Повний клас обчислюваних функцій над номінативними даними

Повний клас обчислюваних (екстенсивних) функцій над номінативними даними складається з усіх функцій, які можна описати через композиції базових функцій і операцій у програмній алгебрі A_{Prog}:contentReference[oaicite:19]index=19. Зокрема, якщо взяти за базу арифметичні та булеві функції ($add, sub, mult, or, neg, eq, gr$, константи, розіменування тощо) та використовувати операції суперпозиції, умовного переходу, циклів і присвоєння, то всі отримані таким чином функції становлять повний клас обчислюваних функцій. Іншими словами, будь-яку функцію з цього класу можна реалізувати як програму в мові SIPL (або еквівалентній) з допомогою наведених операторів і первинних функцій:contentReference[oaicite:20]index=20. Цей факт збігається з концепцією обчислювальної потужності: в межах даного формалізму A_{Prog} отримано всю множину часткових функцій на номінативних даних, які відповідають поняттю «обчислювана функція» в рамках даної системи.

2 Теоретичні основи методів верифікації програм

2.1 Логіка Флойда–Хоара та її аксіоматика

Логіка Флойда–Хоара є формалізмом для доведення властивостей програмних фрагментів у формі трійок Хоара: $\{P\}\ S\ \{Q\}$, де P (передумова) і Q (постумова) — булеві формули над змінними програми, а S — оператор програми. Ідея полягає в тому, що якщо перед виконанням S істина P, а програма S завершується, то після її виконання істинна Q. Логіка визначена через аксіоми і правила виводу:

- **Аксіома пропуску:** $\{P\}$ `skip` $\{P\}$ для будь-якої формули P.
- **Аксіома присвоєння:** для команди $x := E$ справедлива $\{Q[x/E]\}\ x := E\ \{Q\}$, де $Q[x/E]$ — формула Q з підстановкою E замість x.
- **Правило композиції:** якщо виведено $\{P\}\ S_1\ \{R\}$ і $\{R\}\ S_2\ \{Q\}$, то можна вивести $\{P\}\ S_1; S_2\ \{Q\}$.
- **Правило умовного переходу:** якщо $\{P \wedge B\}\ S_t\ \{Q\}$ і $\{P \wedge \neg B\}\ S_f\ \{Q\}$, то $\{P\}$ `if` B `then` S_t `else` $S_f\ \{Q\}$.
- **Правило циклу:** якщо знайдений інваріант I та виведено $\{I \wedge B\}\ S\ \{I\}$, то $\{I\}$ `while` B `do` $S\ \{I \wedge \neg B\}$.

- **Правило наслідку (послідовності):** з $P' \Rightarrow P$, $\{P\}\ S\ \{Q\}$ і $Q \Rightarrow Q'$ випливає $\{P'\}\ S\ \{Q'\}$.

За допомогою цих правил можна поетапно «нарощувати» обґрунтування властивості програми. Наприклад, для циклу необхідно вказати формулу інваріанта I, яка утримується до і після кожної ітерації.

2.2 Коректність числення Флойда–Хоара

Калькуляція Флойда–Хоара є коректною, якщо будь-яка виведена за її правилами трійка Хоара $\{P\}S\{Q\}$ відповідає семантично правильному твердженню про виконання програми. Доведення коректності (тобто теореми зв'язності системи) проводиться індуктивно по правилах:

- Перевіряється, що кожна аксіома в логіці семантично правильна (наприклад, аксіома присвоєння забезпечує, що при виконанні $x := E$ властивість Q після присвоєння виконується точно коли $Q[x/E]$ було істинним перед присвоєнням).
- Доказується, що кожне правило виводу зберігає істинність: якщо припущення правил (попередні трійки) семантично правильні, то і отримана трійка виведення також правильна.

В результаті виходить, що будь-яка виведена за правилами калькуляції трійка дійсно гарантує часткову коректність програми (з точки зору формальної семантики). Іншими словами, якщо $\{P\}S\{Q\}$ доведено в цій логіці, то для всіх початкових станів, що задовольняють P, та за умови завершення S, виконується Q в кінцевому стані. Таким чином, числення Флойда–Хоара є *звуковим*.

2.3 Основні засади методу TLA: досяжність і безпека

Метод TLA (Temporal Logic of Actions) Леслі Лампортом описує поведінку системи за допомогою тимчасової логіки дій. Специфікація системи задається формулою виду

$$Spec \;=\; Init \;\wedge\; \Box[\,Next\,]_{vars} \;\wedge\; L,$$

де $Init$ — початкова умова, $Next$ — формула, що описує один крок переходу (дію), а $\Box$ — оператор завжди (для всіх моментів часу). У цьому формалізмі *безпекові властивості* (safety) перекладаються в інваріанти виду $\Box P$ (наприклад, «ніколи не станеться помилка»), тобто твердження, що певна умова P завжди підтримується. *Досяжність (життєвість)* виражається за допомогою операторів «інколи» ($\Diamond Q$ — зрештою настане Q) або «лідс-до» ($P \rightsquigarrow Q$ — з кожного стану, де P, рано чи пізно настане Q). Таким чином, *досяжність* відповідає гарантії того, що система згодом досягне заданого стану (наприклад, завершення обчислення), а *безпека* — що певне небажане явище ніколи не відбудеться (наприклад, відсутність гонок або порушення інваріантів). У TLA використовують комбінації $\Box$ та $\Diamond$ для виразу таких властивостей. Наприклад, $\Box\neg Error$ означає, що помилка ніколи не настане (safety), а $P \rightsquigarrow \Diamond Q$ — що кожного разу, коли виконується умова P, рано чи пізно виконається умова Q (liveness).

2.4 Принципи верифікації в TLA

Для верифікації системи в TLA слідують таким крокам: спочатку формалізують специфікацію через $Init$ та $Next$. Потім вказують інваріанти Inv та перевіряють, що вони зберігаються при кожному кроці:

$$Init \Rightarrow Inv, \qquad Inv \wedge Next \Rightarrow Inv'.$$

Цим доводять безпеку ($\Box Inv$). Для доведення життєвих властивостей застосовують аргументи індукції по лексикографічних порядках або умов справедливості (fairness). Наприклад, якщо існує функція рангу μ така, що при кожному переході вона незростає і є нижня межа, то це гарантує завершення циклів. У TLA часто використовують техніку «leads-to proof», де розбивають доведення лідс-до на комбінації фактів про $\Box$ та $\Diamond$. Автоматичні засоби перевірки (наприклад, TLC) перебирають усі можливі послідовності станів до заданої глибини, контролюючи, що інваріанти та поставлені лідс-до не порушуються.

2.5 Принципи перевірки моделей у B

Модель у B-методі описується через *абстрактну машину* з визначеним інваріантом та операціями над станом. Принцип перевірки моделі B полягає в побудові і доведенні *обов'язків доказу* (proof obligations) для всіх операцій: необхідно показати, що кожна операція, починаючи з коректного стану (що задовольняє інваріант), призводить до стану, який також задовольняє інваріант. Крім того, перевіряють зв'язок між рівнями абстракції через доказ згортки (refinement): уточнена машина повинна зберігати поведінку абстрактної. Оперативно це часто робиться за допомогою автоматизованих інструментів (Atelier B, Rodin тощо), які перевіряють, що всі інваріантні обов'язки та обов'язки рефінементу виконуються. Крім того, *модельна перевірка* (ProB) може досліджувати всі можливі стани обмеженої системи: перераховуючи всі можливі переходи, вона верифікує, що інваріант справджується у кожному стані і що немає нерелізованих операцій. Таким чином, принцип перевірки в B полягає в суворому відстеженні інваріантів і коректності переходів за формальними правилами рефінементу.

2.6 Властивості B-методу

B-метод забезпечує *згортку за доведенням*: поступове уточнення спек (refinement) з гарантією збереження властивостей. Головними властивостями B-методу є:

- *Інваріантність стану:* кожна машина має інваріант, який перевіряється при всіх операціях. Система гарантує, що цей інваріант не порушується.
- *Коректність рефінементу:* при деталізації (refinement) кожний етап відточення системи вимагає доведення, що нові операції узгоджені з абстрактними. Це забезпечує правильність імплементації відносно специфікації.
- *Точність специфікації:* формальна мова опису (імпертивно-алгебрична) змушує явно вказувати передумови і побічні ефекти операцій, що полегшує аналіз помилок.

Завдяки цим властивостям B-метод вважається «правильним за побудовою» — якщо всі обов'язки виконані, то можна бути впевненим, що програма зберігає задані інваріанти і відповідає специфікації. Також у B-методі інтегровано підтримку генерації коду та перевірки властивостей на рівні моделі, що дозволяє отримати практичні результати верифікації.

2.7 Принципи перевірки моделей у Z

Z-метод описує систему за допомогою *схем* стану і операцій, кожна схема містить декларації змінних та інваріантів, а операції задаються через з'єднання схем. Для верифікації моделі Z також формують обов'язки доказу: необхідно показати, що будь-яка операція, коли застосовується до стану, що задовольняє інваріант, приводить до нового стану, який задовольняє інваріант. Це виконують шляхом аналізу опису змінних в схемах, часто з підтримкою формальних інструментів типу Z/EVES, ZTC чи Community Z Tools. Принцип перевірки полягає в тому, що:

- Ретельно аналізуються пред- і постумови кожної операції, виводяться формули, що перевіряють збереження інваріанта.
- Використовуються теореми множин та предикатів для доведення, що операції є закритими відносно інваріанту.
- При необхідності застосовують символьні перевірки та перевірку типів для виявлення протиріч.

Таким чином, в Z-методі верифікація зосереджується на логічному доведенні властивостей інваріантів і на коректності переходів на рівні специфікації. Ця процедура схожа на B, але в Z акцент на описі структури даних через схеми.

2.8 Властивості Z-методу

Z-метод має такі характерні риси:

- *Абстрактність опису:* Z-нотація використовує теорію множин та предикатну логіку, що робить опис системи високорівневим і незалежним від платформи.
- *Модульність через схеми:* стан та операції чітко розділені на схеми, що полегшує розуміння і перевірку окремих компонентів.
- *Сильне типування та перевірка коректності:* Z-система підтримує потужну механізматику типів і інтерпретує схеми з математичним формалізмом, що знижує ймовірність помилок при написанні моделі.
- *Верифікація на рівні специфікації:* механізми Z/EVES та інші інструменти дозволяють автоматично перевіряти посилання (відсутність суперечностей, коректність інваріантів) на рівні специфікації без необхідності переходу до коду.

Ці властивості роблять Z-метод зручним для формалізації вимог і специфікацій, особливо коли важливо мати строгі гарантії цілісності даних. Водночас Z менше орієнтований на автоматичне доведення (потрібна участь логічних засобів), тому його використовують здебільшого на етапах дизайну та аналізу систем.