УДК 681.3.06

КОНСТРУЮВАННЯ ТА ВЕРИФІКАЦІЯ ПРОГРАМ НА ОСНОВІ СПЕЦИФІКАЦІЙ У КОМПОЗИЦІЙНО-НОМІНАТИВНІЙ МОВІ CNLS

Безверха М.А., Процик П. П.

Національний авіаційний університет,
Київ, проспект Комарова 1, корп. 6,
E-mail: mariya.bezverkha@livenau.net, piter.protsyk@gmail.com

Ця робота продовжує цикл попередніх досліджень, головною метою яких було створення ефективних мов, підходів та програмних засобів розробки якісних програмних систем з використанням формальних методів. На основі побудованих раніше мов специфікації, програмування та математичного апарату представляються практичні підходи автоматизованої побудови та верифікації програм на основі специфікацій. Зокрема, методи перетворення специфікацій у код програми, використання спеціальних засобів мови для автоматичного створення тестових прикладів та можливостей мови по застосуванню методів верифікації під час виконання.

Данная работа продолжает цикл предыдущих исследований, главной целью которых является создание эффективных языков, подходов и программных средств разработки качественных программных систем с использованием формальных методов. На основе построенных языков спецификации, программирование и математического аппарата представляются практические подходы автоматизированной построения и верификации программ на основе спецификаций. В частности, методы преобразования спецификаций в код программы, использование специальных средств языка для автоматического создания тестовых примеров и возможностей языка по применению методов верификации во время исполнения.

This work continues the series of previous studies whose primary purpose was creating efficient languages, approaches and software tools for development of quality software systems using formal methods. On the basis of developed specification and programming languages we present new practical approaches for automated construction and verification of programs based on formal specifications. In particular, methods of conversion specifications in the program code, the use of special means of language to automatically create test cases and language capabilities for the application of run-time verification.

Вступ

Процес верифікації – це важливий етап розробки програмних систем. Метою процесу верифікації є встановлення того, що поведінка програми співпадає з очікуваною поведінкою. Це інтуїтивно зрозуміле визначення не уточнює, що таке виконання програми, її поведінка – реальна та очікувана, і як впевнитись, що вони співпадають. В залежності від обраних визначень процес верифікації буде суттєво різним. Так, при ручній верифікації (тестуванні), під поведінкою найчастіше розуміють те, що користувач бачить на екрані при роботі програми разом з результатами, які вона виробляє: файли, друковані документи тощо. Часто очікувану поведінку програми описують лише неформально у вигляді результатів її виконання та проміжних станів.

На практиці використання формальних, тобто заснованих на математичних методах, засобів верифікації є досить складним [1]. Однією з причин такого стану справ – є недостатній рівень автоматизації формальних методів, складність їх інтеграції в існуючі процеси.

Ця робота продовжує цикл досліджень [2, 3, 4], головною метою яких було створення ефективних мов, підходів та програмних засобів розробки якісних програмних систем з використанням формальних методів. На основі побудованих у попередніх роботах мов специфікації [2,4], програмування та математичного апарату представляються практичні підходи автоматизованої побудови та верифікації програм на основі специфікацій. Зокрема, методи перетворення специфікацій у код програми, використання спеціальних засобів мови для автоматичного створення тестових прикладів та можливостей мови по застосуванню методів верифікації під час виконання.

У проведених дослідженнях широко застосовуються принципи та математичний апарат композиційно-номінативного підходу [5]. Визначення використаних у статті понять були взяті з роботи [6].

Специфікація програм

Верифікації програмних системи не можлива без чіткого опису очікуваної поведінки системи, що витікає з мети цього процесу. Такий опис поведінки може виконуватись різними способами – за допомогою вимог, тестових сценаріїв (*test cases*), формальних специфікацій тощо.

У контексті даної роботи для специфікації поведінки буде використовуватись підхід, що ґрунтується на використанні мови *Z-Notation* [7] та композиційно-номінативної семантики, які запропоновані у роботах [3, 4]. У рамках цього підходу система описується в термінах станів та переходів з одного стану в інший. Функціонування системи починається з деякого початкового стану та продовжується (можливо, нескінченно довго), поки існує можливість переходу в інший стан. Іноді виділяють множину станів, яку називають заключною, і при переході системи в один з таких станів її виконання припиняється. Кожна програмна система повинна мати хоча б один початковий стан. Тому для кожної специфікації повинна виконуватись наступне твердження

$$\exists StateSpace \bullet InitialState .$$

Специфікація програмної системи у *Z-Notation* являє собою набір схем разом з тестовими поясненнями. Схеми описують поведінку та властивості системи. Схема у *Z-Notation* – це засіб структурування специфікації. Схема складається з декларативної та предикативної частин. У декларативній частині описуються змінні, які приймають участь у схемі. Предикативна частина накладає на змінні обмеження, описує їх властивості, встановлює взаємовідношення між ними у формі предикатів логіки першого порядку. Існує три типи змінних – вхідні, вихідні та змінні стану системи. Синтаксично схема має такий вигляд: [*Declaration* | *Predicates*].

У специфікаціях можна використовувати схеми у наступних випадках: для завдання типів даних, у визначеннях, у предикатах.

У загальному випадку схема має таку структуру:

$\boldsymbol{Scheme}$ $= [x_1 : \boldsymbol{S_1} ; \ldots ; x_n : \boldsymbol{S_n};$

$x'_1 : \boldsymbol{S_1} ; \ldots ; x'_n : \boldsymbol{S_n};$

$i_1? : \boldsymbol{T_1} ; \ldots ; i_m? : \boldsymbol{T_m};$

$o_1! : \boldsymbol{U_1} ; \ldots ; o_p! : \boldsymbol{U_p}$ |

$Pre_k (i_1 , \ldots , i_m , x_1 , \ldots , x_n);$

$Inv_l (x_1 , \ldots , x_n); Inv_o (x'_1 , \ldots , x'_n);$

$Op_j (i_1, \ldots, i_m, x_1, \ldots, x_n, x'_1, \ldots, x'_n , o_1, \ldots, o_p)],$

де
$x_1 : \boldsymbol{S_1} ; \ldots ; x_n : \boldsymbol{S_n}$ – змінні стану до виконання операції,
$x'_1 : \boldsymbol{S_1} ; \ldots ; x'_n : \boldsymbol{S_n}$ – змінні стану після виконання операції,
$i_1? : \boldsymbol{T_1} ; \ldots ; i_m? : \boldsymbol{T_m}$ – вхідні змінні,
$o_1! : \boldsymbol{U_1} ; \ldots ; o_p! : \boldsymbol{U_p}$ – вихідні змінні,
Pre_k – предикати, які задають передумови операції,
Inv_l, Inv_o – інваріантні відношення, які накладаються на змінні,
Op_j – предикати, які задають операцію, тобто пов'язують поточний та наступний стан системи

На практиці використовуються різні рівні деталізації поняття стану та переходу. Будемо представляти стани системи за допомогою номінативної множини [5,6]. Вона складається з компонентів вигляду *ім'я→значення*. Переходи тлумачаться як номінативні операції, що відображають одні стани системи у інші. Схеми поділяються на три типи – схеми станів, схеми ініціалізації та схеми операцій

$SchemaText = StateSchemaText \mid InitSchemaText \mid OpSchemaText;$

$StateSchemaText = [x_1 , \ldots , x_n \mid P(x_1 , \ldots , x_n)];$

$InitSchemaText = [x_1' , \ldots , x_n' \mid Init(x_1' , \ldots , x_n')];$

$OpSchemaText = [x_1, \ldots, x_n, x_1', \ldots, x_n' \mid Pre(x_1 , \ldots , x_n); Op(x_1 , \ldots , x_n, x_1' , \ldots , x_n'); Post(x_1' , \ldots , x_n')].$

Зафіксуємо множину доступних імен

$Names == \{x_1 , \ldots , x_m\}.$

По мірі необхідності множину імен можна буде поповнювати. Але вона завжди буде містити лише скінчену кількість елементів. Достатньо розглядати схеми у такій синтаксичній формі

$S = [x_1, \ldots, x_n, x_1', \ldots, x_n' \mid P(x_1 , \ldots , x_n), Q(x_1 , \ldots , x_n, x_1' , \ldots , x_n'), R(x_1' , \ldots , x_n')].$

Тоді введемо такі позначення $Names(S) = \{ x_1, \ldots, x_n \}$ та $Names'(S) = \{ x_1', \ldots, x_n' \}$ відповідно.

Тепер можемо надати формальне визначення специфікації. Отже, синтаксичною специфікацією програмної системи у *Z* назвемо впорядкований текст, що складається з довільного скінченого числа схем станів $ST_1, \ldots, ST_k$, однієї або декількох схем ініціалізації $I_1, \ldots, I_m$, довільного скінченого числа схем операцій $Op_1, \ldots, Op_n$ та довільного числа допоміжних схем-визначень $C_1, \ldots C_p$. Схеми визначення вводяться як синтаксичні скорочення. При цьому, при переході до семантики, вони об'єднуються з семантикою схеми, в

якій вони використовуються. Схеми можна синтаксично об'єднувати за допомогою операції синтаксичного об'єднання схем.

Як було показано у [4], кожна схема *Sch* буде визначати клас номінативних множин ***S(Sch)*** такий, що кожен елемент цього класу буде задовольняти умовам специфікованим у відповідній схемі. Детальні викладки семантики схем були представлені у роботах [3,4].

Враховуючи попередні міркування, специфікацією програмної системи у *Z-Notation* будемо називати наступну четвірку *Spec=(ST,I,OP,C)*, де $ST=\{ ST_1,...,ST_k \}$ – множина схем станів; $I=\{ I_1,...,I_m \}$ – множина схем ініціалізації; $OP=\{Op_1,...,Op_n \}$ – множина схем операцій; $C=\{ C_1,...C_p \}$ – множина допоміжних схем.

Моделлю *Z*-специфікації, (або моделлю, яку індукує специфікація) будемо називати трійку, що визначається наступним чином

$$Model = \begin{pmatrix} S(ST_1 \cup ST_2 \cup ... \cup ST_k); \\ S(I_1 \cup I_2 \cup ... \cup I_m); \\ S(Op_1) \cup S(Op_2) \cup ... \cup S(Op_n) \end{pmatrix} = (ST, I, OP).$$

Відзначимо, що *OP* фактично задає часткову багатозначну функцію з множини усіх номінативних даних *ND* у *ND*.

Зрозуміло, що не кожна специфікація визначає програмну систему здатну щось обчислювати. Для цього необхідно (але не достатньо), щоб отримана модель системи задовольняла наступним умовам:

1. $ST \neq \varnothing$, тобто простір станів не вироджений;
2. $I \subseteq ST$ та $I \neq \varnothing$, тобто існує хоча б один початковий стан;
3. $\forall st \in ST : OP(st)\downarrow \Rightarrow Op(st) \subseteq ST$, тобто операції системи не виходять за межі простору станів.

Стан *st* системи називається заключним, якщо $OP(st)\uparrow$.

З наведених визначень випливає, що кожна коректна *Z* специфікація задає транзиційну систему. Ця транзиційна система і буде описувати поведінку досліджуваної програмної системи. При чому, на кожному кроці виконання вона буде задовольняти умовам специфікації.

Отже, наступним кроком, у роботі буде досліджуватись перехід від цієї формальної моделі до коду програми та побудові засобів її верифікації.

Математичний інструментарій мови CNLS

Перейдемо до розгляду мови програмування, яка буде використовуватись для досягнення поставленої мети. Ця мова була вперше представлена у роботі [2] і з того часу зазнала активного розвитку. Згодом її назва була змінена на «композиційно-номінативну мову специфікації програмних систем» (*Composition-Nominative Language for program Specification)* або скорочено *CNLS.*

Найбільший інтерес для даної роботи буде являти так званий математичний інструментарій мови. Тобто формалізми, запозичені з математики та розвинуті, враховуючи потреби програмування [5].

Множини

Поняття множини широко використовується у мові *CNLS*. Основний тип даних у мові – номінативний об'єкт, який на семантичному рівні моделюється номінативною множиною. Окрім пар «ім'я»-«значення», номінативний об'єкт містить пари спеціального вигляду – «ім'я»-«функція», що не суперечить загальному визначенню номінативної множини, але суттєво розширює семантику мови. У мові кожна обчислювана функція може розглядатись як дане і як функція. Можливість тлумачення функції як даного важлива для побудови ефективних методів верифікації.

Математичний інструментарій мови *CNLS* дає можливість оперувати наступними, дещо умовними, класами множин

- Множини чисел: ***N, Z, D;***
- Мовні множини: множини слів деякого алфавіту, множини, що задаються регулярними виразами;
- Номінативні множини: множини пар ім'я-значення *ND = [v->1, y->[z->4]]*;
- Скінчені множини деяких об'єктів;
- Множини, що задаються предикатом належності, наприклад, $Odd = \{x \in N \mid x = 2k+1, k = 0 \wedge k \in N\}$;
- Типи даних: кожен тип даних задає множину об'єктів (клас).

Для множин визначені класичні теоретико-множинні операції: об'єднання, перетину, різниці тощо.

Функції

На семантичному рівні програма у мові CNLS трактується як функція над номінативним даним (станом), побудована з більш простих функцій за допомогою композицій, та відображає вхідні дані програми у вихідні. Результат виконання програми може бути невизначеним або символічно – $\perp$. Тому, кожна програма в загальному випадку задає часткову функцію над множиною номінативних даних.

Кожна синтаксична конструкція визначає номінативну функцію певного рівня, що належить класу усіх відображень вигляду $ND \to ND$.

Хоча зрозуміло, що всі функції на рівні семантики – квазіарні [6], для зручності будемо допускати такі синтаксичні скорочення для позначення застосування функції до відповідного номінативного даного [8]. Або, що те саме, але більш звично для програмування – виклику тіла функції з параметром. Тут символьний запис $t_1 \sim t_2$ означає, що «терм t_1 є скороченим записом семантичного терму t_2»:

- $f_{st}() \sim f(st)$;
- $f_{st}(v_1) \sim f(st \nabla [v_1 \to x_1])$, *для деякого значення параметру* x_1;
- $f_{st}(v_1,...,v_n) \sim f(st \nabla [v_1 \to x_1, ..., v_n \to x_n])$, *для деяких значень параметрів* $x_1, ..., x_n$.

Найпростішими є константні функції

- *відсутнє значення:* $Null \sim null() \equiv null$;
- *константа нуль:* $0 \sim \Rightarrow 0 \equiv 0$;
- *числові константи:* $n \sim \Rightarrow n \equiv n, \forall n \in N$, *або* Z, *або* R.

Зауважимо, що числові проміжки, які є областю значення числових констант є, як правило, обмеженими і скінченими (навіть для дійсних чисел) в силу об'єктивних причин, що накладаються можливостями обчислювальних пристроїв.

Запис *sem(t)* буде позначати семантичний об'єкт, що відповідає синтаксичному терму *t*.

Розглянемо синтаксичне визначення функції, основного об'єкту мови *CNLS,* за допомогою форм Бекуса-Наура

functionDefinition *:=* $f(v_1, ..., v_n)$ $global(a_1, ..., a_m)${***functionBody***}
functionBody *:=* ***functionCall ';' functionBody*** | *empty*
functionCall := *id* ***'(' functionCall ,…, functionCall ')'.***

Як бачимо, визначення складається з

- імені – *f;*
- списку обов'язкових імен-аргументів - $v_1, ..., v_n$;
- списку змінних стану, які функція може змінити - $a_1, ..., a_n$;
- та визначення тіла функції ***functionBody***, що являє собою послідовність викликів функцій.

Спочатку визначимо семантику для виклику тіла (обчисленні значення) функції над заданими значеннями параметрів. Нехай існує визначення функції у формі $f(v_1, ..., v_n)$ $global(a_1, ..., a_m)$ { ***functionBody*** }. Тоді терм $f(x_1, ..., x_n)$ позначає результат обчислення функції f на даному $st \nabla [v_1 \to x_1, ..., v_n \to x_n]$. Цей результат отримується послідовним виконанням обчислень заданих термом ***functionBody***

$f(v_1, ..., v_n)$
global$(a_1, ..., a_m)$ {
$g_1(v_1, ..., v_n, a_1, ..., a_m)$;
$g_2(v_1, ..., v_n, a_1, ..., a_m)$;
. . .
$g_k(v_1, ..., v_n, a_1, ..., a_m)$;
}

$t = f(x_1, ..., x_n)$.

Тоді $sem(t) = state \nabla [result \to (result \Rightarrow body), a_1 \to (a_1 \Rightarrow body), ..., a_m \to (a_m \Rightarrow body)]$ при $body = sem(g_1(state')) \circ ... \circ sem(g_k(state'))$ та $state' = state \nabla [v_1 \to x_1, ..., v_n \to x_n]$.

Розглянемо представлення базових номінативно обчислюваних функцій у CNLS:

- константа нуль $\Rightarrow 0$, – *0;*
- константа один $\Rightarrow 1$, – *1;*
- порожнє дане $\bar{[]}_D$ – *empty;*
- функція вибору іменованої компоненти $\mathbf{\Lambda}_D$ – *d.v*;
- бінарна композиція об'єднання $\cup_D(d_1,d_2)$ – (d_1+d_2);
- бінарна композиція віднімання $\setminus_D(d_1,d_2)$ – (d_1-d_2);
- бінарна композиція рівності $(=_W)_D(d_1,d_2)$ – $(d_1 == d_2)$;
- предикат належності $(\in W_D)(a, d_1)$ – $(d_1.Has(a))$;

- існування імені *(!v)(d) – (d.HasName(v));*
- бінарна композиція присвоювання (іменування) $as_D(f,g)(d) – (empty.[\, f\,(d) \rightarrow g(d)]);$
- взяття значення (роз'іменування) $cn_D(f,g)(d) – (g(d)[f(d)]);$
- перейменування (реномінації) $R_D(a,b)(d) – ([b \rightarrow d.a] + (d – [a \rightarrow d.a]))$.

Синтаксичній спектр функцій мови значно ширший за набір базових, але на семантичному рівні вони можуть бути зведеними до вказаних базових.

Функції з перевірками

Мова володіє потужним механізмом, який дозволяє в декларативній формі накладати умови на функції і на програму. Ці специфікації мають форму перед-, після- та інваріантних умов.

Крім того, існує можливість використовувати булеві вирази (предикати), які будуть обчислюватись під час виконання програми, що забезпечить додатковий контроль при розробці, тестуванні та використанні програми.

Синтаксично функція з перевірками задається наступним чином

function *name(...)* ***global*** *(...)*
[***pre****(boolExpr$_1$);* ***post****(boolExpr$_2$);* ***inv****(boolExpr$_3$);]*
{
Body*;*
}.

Вираз, заданий як передумова (*pre*), обчислюється над станом безпосередньо перед початком обчислення тіла функції. Відповідно вираз, заданий у післяумові (*post*), обчислюється одразу після тіла функції. Інваріантні умови (*inv*) обчислюються кожен раз при зміні значень у стані та одразу після обчислення виразу передумови. Обчислення умов не повинно змінювати значень стану.

У роботі [8] подібні механізми накладання умов на функцію носять назву контракту. Існують мови, у яких в тій чи іншій мірі реалізується цей механізм *Eiffel* [9], *Spec#* [10] та інші.

Звичайно, що при написанні програми, у контракті можна використовувати тільки обчислювані предикати. У CNLS до них відносяться такі:

- предикати номінативної належності;
- предикат існування імені у даному;
- логіко-функціональні предикати: порівняння, рівності, порожності даного;
- агрегатні предикати: *any(d,p)* – хоча б один, *all(d,p)* – кожен, та їх похідні;
- усі булеві функції мови (функції, що мають областю значень множину {T,F}).

Концепція контракту, окрім суто практичної користі, відіграє суттєву методологічну роль у сучасному програмуванні [8-10]. За допомогою контрактів можна декларативно

- фіксувати припущення розробника цього коду, щодо вхідних, вихідних даних та перетвореннях стану в процесі обчислення;
- формувати базис тверджень про програму для формального аналізу її поведінки під час виконання;
- утворювати зв'язок між програмою та її специфікацією.

Розглянемо реалізацію концепції контракту у мові CNLS. При переході до семантики функції будемо використовувати такі визначення

$$sem(f(x_1,...,x_n)) = \begin{cases} state\nabla[result \rightarrow (result \Rightarrow body)], \text{якщо виконуються співвідношення (1)} \\ невизначено, \text{інакше.} \end{cases}$$

$$\begin{cases} result \Rightarrow sem(boolExpr_1(state)) = true, \\ body(state)\downarrow; \\ state' = state\nabla[result \rightarrow (result \Rightarrow body(state))]; \\ result \Rightarrow sem(boolExpr_2(state')) = true. \end{cases} \quad (1)$$

При цьому, семантика операції присвоєння зміниться наступним чином

$$sem(v = f(x_1,...,x_n)) = \begin{cases} state'\nabla[v \rightarrow (result \Rightarrow state')], \text{якщо виконуються співвідношення (2)} \\ невизначено, \text{інакше.} \end{cases}$$

$$\begin{cases} state' = state \nabla sem(f(x_1,...,x_n)); \\ sem(f(x_1,...,x_n))\downarrow; \\ state'' = state' \nabla [v \to (result \Rightarrow state')]; \\ boolExpr_3(state) = boolExpr_3(state'') = true. \end{cases} \quad (2)$$

Враховуючи представлену семантику *Z-Notation*, можна провести аналогію між визначенням операцій за допомогою схем та функціями з перевірками у *CNLS*: у рамках семантики Хоара/Мейера схеми операцій визначають контракт методу.

В цьому контексті цілком природною і практично обумовленою постає задача побудови процедури перетворення специфікації з *Z-Notation* у шаблон виконуваною програми *CNLS*. При цьому, за рахунок контрактів досягається узгодженість програми з специфікацією.

Предикати

Предикати – це вирази, які приймають значення з двозначної множини, наприклад {істина, фальш}. У випадку часткових предикатів враховується також можливість невизначеного значення. У своїй статті [11] А. Мартін окреслює проблему невизначеності в контексті семантики *Z-Notation*, як одну із складних проблем, з якою стикаються дослідники. Тут буде розглядатись випадок, коли невизначеність вводиться як деяке спеціальне значення, скажімо *undef.*

У *CNLS* є такі оператори булевої алгебри: *and (&)*, *or (|)*, *not (!)*. Оператори приналежності до множини – *in*, до типу – *is*. Крім того, предикатом буде також функція, яка повертає значення *true* (істина) або *false*(фальш). Функцію можна додатково позначати як предикат за допомогою такого контракту: *function pred(x,y,…) [pre(); post(**result** is Boolean); invariant(unchanged(state, x,y,..,))] {…}*. Тут *unchanged(…)* – системний предикат, який вказує на те, що обчислення тіла функції не призводить до змін у стані змінних.

Якщо при обчисленні предикату виникає невизначена ситуація, наприклад один з аргументів є невизначеним, то значення предикату автоматично приймається невизначеним. Крім того, результат виконання програми у такому випадку теж буде невизначеним.

Перетворення специфікації у CNLS програму

При використанні традиційних мов програмування розв'язання задачі перетворення специфікації у виконуваний код є нетривіальною проблемою. Крім того, результуючий код програми слабко зв'язаний з початковою специфікацією і можливість відслідковувати обернені зв'язки та виконувати взаємні корективи (специфікації-коду) втрачається.

Проте існують спеціалізовані мови програмування, такі як *Spec#, Eiffel* та інші, які за рахунок розширених мовних конструкцій дозволяють зберігати зв'язок між кодом програми та специфікацією і далі використовувати ці зв'язки при формальному аналізі програми. Наприклад, у роботі [12] представлена процедура побудови програми *Spec#* на основі специфікації. До мов такого класу відноситься і CNLS.

Опишемо процес, який дозволяє автоматизовано перетворювати коректні специфікації у формі *(ST,I,OP,C)* в програми на мові CNSL. Специфікації може відповідати багато синтаксично різних програм, проте поведінка кожної з них буде збігатися з властивостям, заданими у специфікації.

Процес перетворення у виконуваний код відбувається поступово, шляхом послідовних кроків уточнення. Перший крок – створення початкової програми може виконуватись автоматично. При цьому, змінні описані у схемах станів (*ST),* перетворяться у змінні стану програми. Вхідні дані та початковий стан програми повинні задовольняти умовам схем ініціалізації *(I)*. Без втрати загальності можна вважати, що множини *ST* та *I* є одноелементними. Якщо це не так, то їх можна отримати шляхом застосування операції об'єднання схем по всім елементам множин. Кожна схема з множини *OP* визначає функцію на станах системи.

При цьому структурно програма буде складатись з функцій *Init(), Inv(st)*, $Op_1(st)…Op_n(st)$. Припустимо, що $S = [x_1 ,..., x_n \mid P(x_1 ,..., x_n)]$; $I = [x_1' ,..., x_n' \mid I(x_1' ,..., x_n')]$. Тоді функція *Init(),* що задає початковий стан, будується за схемою *I* наступним чином

***function** Init() [pre(empty(st)); post(pred_I(**result**)); invariant(Inv(st))] {*
  st = [
    $x_1 \to null,\ x_2 \to null,\ \ldots\ x_n \to null];$
}.

Предикат, що задає післяумови у контракті *pred_I,* будується на основі визначення $I(x_1' ,..., x_n')$ у схемі. При чому можливі два випадки, перший, коли визначення $I(x_1' ,..., x_n')$ задає вираз, що можна автоматично перетворити у вираз мови та більш складний, коли автоматичне перетворення не можливе. У другому випадку, будується пуста функція *pred_I,* яка повинна обчислювати значення предикату. Розробка алгоритму обчислення, в цьому випадку, віддається на розсуд розробника, так само, як і тіла функції *Init.*

Функція, що обчислює значення предикату *Inv,* будується на основі визначення схеми *S* таким чином

function *Inv(st)[pre(); post(**result** is Boolean); inv(unchanged(st))]{*
return (Names(st).Subset([x_1 ,..., x_n])) && pred_P(st);
}.

При цьому, ситуація з функцією *pred_P*, яка обчислює значення предикату, аналогічна попередньому випадку. Для кожної схеми, що задає операцію *Op = [x_1,...,x_n,x_1',...,x_n' | Pre(x_1 ,..., x_n);Op(x_1 ,..., x_n, x_1' ,..., x_n'); Post(x_1' ,..., x_n')],* будується окрема функція програми:

function *Op(st)[pre(pred_Pre(st)); post(pred_Post(**result**)&pred_Op(st, result)); invariant(Inv(st))]{*
throw *NotImplementedException();*
}.

Розглянемо типовий приклад схеми, що задає операцію та функцію, яка може бути отримана за нею

- схема *f = [x:Z, x':Z| x>0; x' mod 2 == 0],*
- функція *function f(st) [pre(st(x)>0); post(result(x) % 2 == 0); invariant(st(x) is int);] { обчислення }.*

Отримана за специфікацією програма не містить тексту обчислення, вона є шаблоном для подальшого уточнення розробником. Важливість цього шаблону полягає в тому, що він дозволяє в подальшому будувати програми, які задовольняють вимогам, накладеним специфікацією.

Наступні кроки уточнення творчі. Вони виконуються розробником системи та полягають у реалізації виконуваної частини програми, яку неможливо отримати з декларативних специфікацій на першому кроці. Так, за допомогою предикатів, можна виразити складні твердження про властивості функції. Задати обчислення цієї функції можна різними способами – це і є задача розробника.

Створена таким чином програма, відкриває ряд нових можливостей для верифікації, серед яких

- застосовування засобів верифікації, характерних для *Z-Notation*, коли властивості програми будуть відповідати властивостям специфікації за рахунок контрактів, з одним обмеженням, – контракт програми є істинним;
- автоматизована генерація тестових прикладів та їх виконання;
- використання методик верифікації під час виконання.

Автоматизоване тестування CNLS програм

Розглянемо процес побудови тестових прикладів та верифікації програм, створених за представленою методикою. Завдяки тісному зв'язку отриманої програми з формальною специфікацією, цей процес добре піддається автоматизації. Це робить його також потужним засобом тестування.

Розглянемо функцію з попереднього прикладу

```
function f(x) [ pre(x>0); post(x % 2 == 0); invariant(x is int); ] {
   x = x*4 ;
  return x;
}.
```

Умови, що накладаються за допомогою контракту, повинні виділяти серед усіх даних ті, які належать області визначеності та області значень функції. Цю інформацію можна використовувати для побудови прикладів, які б задовольняли накладеним умовам. Існує спеціальний клас програмних засобів, здатних виконувати цю задачу – це *Satisfiability Modulo Theory (SMT)* солвери, серед яких – *Yices*, *Z3* [13, 14].

Покажемо на прикладі солверу *Yices*, яким чином можна використовувати контракти для побудови тестових даних. Для цього перетворимо умови контракту в предикати, записані на мові солвера

Контракт *CNLS*	***Yices***
invariant(x is int)	*(define f1::(-> int int))* *(define x::int)*
pre(x>0);	*(assert (> x 0))*
post(x % 2 == 0);	*(assert (= (mod (f1 x) 2) 0)).*

Отримаємо такий вхідний текст

```
(define x::int)
(define f::(-> int int))
(assert (> x 0))
(assert (= (mod (f x) 2) 0))
(check).
```

Розглянемо протокол роботи програми після запуску з такими параметрами

yices.exe -e 1.txt

Результат	**Опис**
Sat	Тестовий приклад знайдено
(= x 1)	При *x = 1*
(= (f x) 0)	якщо *f(x) = 0* тоді
(= (mod 0 2) 0)	умова контракту x%2 == 0 задовольняється.

Цей результат було отримано лише на основі предикатів з контракту, тому значення функції *f(x) = 0* може відрізнятись від реального значення функції на цьому даному. Поки що це не є важливим, оскільки мета кроку – отримати вхідні дані, які б задовольнили умови контракту. Далі виконується обчислення функції *f(x)* при *x=1*. Отримаємо такий результат *f([x→1]) = x⇒ *4 = 4.*

Тепер модифікуємо текст для солвера враховуючи нову інформацію

```
(define x::int)
(define f1::(-> int int))
(assert (> x 0))
(assert (= (mod (f1 x) 2) 0))
(check)
(assert (= x 1))
(assert (= (f1 x) 4))
(check).
```

Після запуску буде одержано такий результат:

```
sat
(= x 1)
(= (f1 1) 4)
(= (mod 4 2) 0)
```

Тобто, отримане в результаті обчислень функції значення (стан системи) теж задовольняє умовам контракту. А отже, функція на цьому конкретному даному задовольняє умови початкової специфікації, і її поведінку можна вважати такою, що не суперечить вимогам.

Метод побудови тестових прикладів складається з таких кроків.

Крок 1. Використовуючи умови контракту, побудувати набір даних – d_1, що буде його задовольняти.

Крок 2. Отриманий набір даних підставити на вхід функції, та обчислити її значення – y_1. Якщо значення функції невизначене (функція зациклилась, виникла виключна ситуація, тощо), тоді вважаємо, що програма обчислення функції містить помилку, і необхідно або посилити умови у специфікації, або виправити помилку та перейти до першого кроку.

Крок 3. До умов, що були побудовані на першому кроці, додаємо нове твердження: $f(d_1) = y_1$. Та перевіряємо, що отримана система тверджень не містить суперечностей. Якщо суперечність існує, тоді модифікуємо код програми або її специфікації.

Крок 4. Далі продовжуємо процедуру пошуку нових тестових даних, додаючи на кожному кроці умову $x \neq x_t$ або *(assert (/= x xt))*, де *xt* – значення змінної отримане на попередньому кроці. Та переходимо до другого кроку.

При цьому, якщо на деякому кроці процедура призведе до негативного результату на етапі перевірки значення функції, це одразу означає, що програма, яка обчислює значення функції не відповідає умовам специфікації, а отже містить помилку.

За рахунок автоматизації цю процедуру можна використовувати для перевірки відповідності реалізації початковій специфікації на великих об’ємах даних, тим самим суттєво зменшуючи ризик появи дефектів у коді.

Верифікація під час виконання

Верифікація програми в процесі виконання (або *Run-time* верифікація) [15] – це підхід, при якому тестування вхідної програми на відповідність специфікації відбувається власними засобами програми під час її виконання з використанням формальних підходів. *Run-time* верифікація виступає в якості додаткового засобу, що гарантує коректність програми. В якості окремого випадку верифікації під час виконання можна розглядати модульне тестування (*UnitTesting*), яке полягає у виконанні тестових прикладів та порівнянні результатів з еталонними даними. Недоліком модульного тестування є те, що і тестові приклади, і еталонні результати створюються програмістами на основі власних міркувань, які не є формальними. Тому вважається, що верифікація в процесі виконання є більш надійним засобом гарантування коректності програми ніж тестування (включаючи модульне тестування), та менш надійним ніж формальні методи [15].

При запропонованому підході створення програми із формальної специфікації, контракти виступають одним з ключових елементів верифікації під час виконання. Кожен раз при обчислені функції або модифікації змінних стану програма перевіряє валідність цих дій шляхом обчислення значень предикатів. Крім цього,

система веде протокол усього процесу обчислення, який при порушенні умов контракту можна використовувати для аналізу та налагодження програми.

Розглянемо наступний приклад. Нехай потрібно розробити функцію, яка з масиву натуральних чисел виділяє ті, що повторюються. Цей приклад носить лише демонстраційний характер, але він є достатньо показовим.

Розпочнемо із *Z* специфікації. Простір станів буде складатись із двох множин. Множини чисел, що не повторюються - *Unique*, та тих, що повторюються – *Repeated*. Очевидно, вони не повинні перетинатися.

$$StateSpace = [Unique : P(N), Repeated : P(N) \mid Unique \cap Repeated = \{\}].$$

При додавання нового елементу до стану, він буде спочатку долучатися до множини *Unique* елементів, а при повторному додаванні вилучатись з цієї множини і долучатись до множини *Repeated*:

$$AddItem(x) = [\ Unique, Repeated : P(\mathrm{N}); Unique', Repeated' : P(\mathrm{N}); x? : N \mid$$
$$x \notin Unique \Rightarrow Unique' = Unique \cup \{x\};$$
$$x \in Unique \Rightarrow Unique' = Unique \setminus \{x\} \wedge Repeated' = Repeated \cup \{x\}]$$

Відповідна програма буде складатись із двох частин: блоку ініціалізації та функції додавання нового елементу.

Блок ініціалізації

```
Unique = new Set();
Repeated = new Set();.
```

Функція додавання елементу

```
function AddItem(x)
[ pre(Naturals.Contains(x)); // відповідає x ∈ N
post(Unique.Contains(x)|Repeated.Contains(x)); // відповідає x ∈ Unique ∧ x ∈ Repeated
invariant(Unique.Intersect(Repeated)==Sets.Empty); // відповідає Unique ∩ Repeated = {}
]
{ if (!Unique.Contains(x)) Unique.Add(x);
  else { Unique.Remove(x); Repeated.Add(x); }
}.
```

У прикладі варто звернути увагу на блок інваріанту, де специфікуються умови порожності перетину множин (з умови схеми *StateSpace*), а також післяумову, яка гарантує, що елемент буде долучено хоча б в одну множину.

На рис. 1 зображено обробку описаного прикладу в системі *CNLS*:

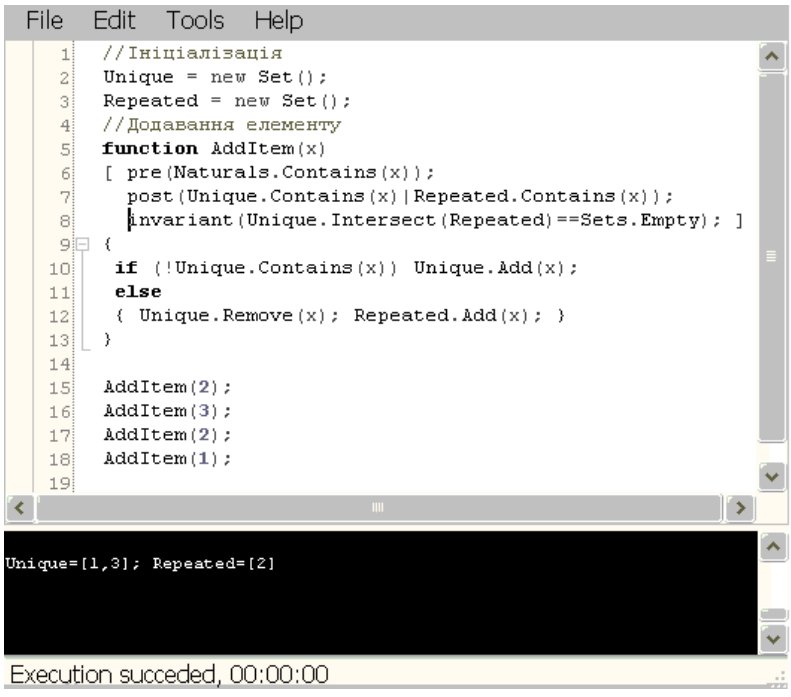

Рис. 1. Унікальні та повторювані елементи

Відзначимо також, що зміна порядку операцій вилучення та додавання елементу у множини призведе до порушення умови інваріанту: зміна *Unique.Remove(x); Repeated.Add(x);* на *Repeated.Add(x); Unique.Remove(x).* Від моменту додавання елемента до множини *Repeated* і до моменту його вилучення із множини *Unique,* ці дві множини будуть мати непорожній перетин. І, хоча результат виконання функції від цього не зміниться, такий стан програми може призводити до небажаних сторонніх ефектів, наприклад у випадку багатопоточності.

Це зауваження показує, що навіть достатньо тривіальні приклади програм, можуть містити в собі важко передбачувані аспекти, які легко ідентифікуються при використанні формальних підходів. Все це говорить на користь запропонованого методу, хоча його використання і не гарантує відсутності інших помилок. В цілому, запропонований підхід може застосовуватись як додатковий засіб перевірки часткової коректності програм, та бути додатковим «ременем безпеки» при розробці програмного забезпечення.

Висновки

Результатом роботи є розвиток підходів автоматизованої верифікації програм, створених за допомогою мови програмування CNLS. Були запропоновані методи перетворення формальних специфікацій у код програми. Розроблено методи побудови тестових прикладів за допомогою солверів. Представлено засоби, що підтримають методи верифікації під час виконання.

Важливість отриманих результатів полягає в їх практичній значимості. Кожен з представлених методів добре підлягає автоматизації та може використовуватись при розробці програмних систем. Крім того, в рамках цієї роботи були розроблені діючі прототипи для обробки представлених мов специфікації та програмування.

Література

1. J. C. Knight, C. L. Dejong, M. S. Gibble, L. G. Nakano Why Are Formal Methods Not Used More Widely // Fourth NASA Formal Methods Workshop. – 1997, P. 1-12.
2. Процик П.П., Композиційно-номінативна мова програмування Script.NET // “Проблеми програмування”. № 2-3 (спеціальний випуск). Київ – 2008, –С. 323-331.
3. Процик П.П. Композиційно–номінативний підхід до побудови семантики Z–Notation // Вісник київського університету. Сер.: Фізико-математичні науки. – 2008. – №1. –С. 116–120.
4. Процик П.П. Композиційно–номінативний підхід до специфікації програмних систем у мові Z–Notation // Вісник Київського університету. Сер.: Фізіко-математичні науки.–2009.–№1., –С. 133-138.
5. Никитченко Н.С. Композиционно-номинативный подход к уточнению понятия программы. // Проблемы программирования. – 1999.–№1. С. 16– 31.
6. Нікітченко Н.С., Шкільняк С.С., Математична логіка та теорія алгоритмів:підручник – К.: Видавничо-поліграфічний центр «Київський університет», 2008. – 528 с.
7. Jim Woodcock, Jim Davies: Using Z. Specification, Refinement, and Proof // Prentice Hall, New York– 1996. – 523 p.
8. B. Meyer Object-oriented software construction – New York: Prentice-Hall International Series of Computer Science, 2000. – 1296 p.
9. B. Meyer Eiffel: The Language– New York: Prentice-Hall International Series of Computer Science, 1992. – 296 p.
10. M. Barnett, K. Rustan, M. Leino, W. Schulte, The Spec# programming system: An Overview // CASSIS 2004 Proceedings – 2004. P. 29-69.
11. Martin Andrew Relating Z and First-Order Logic // Journal of Formal Methods. – 1999. – Volume II. – P. 1266-1280.
12. Shengchao Qin; Guanhua He, Linking Object-Z with Spec# //Proc. of 12th IEEE International Conference on Engineering Complex Computer Systems, 11-14 July 2007, P. 185 – 196.
13. L. Moura N. Bjørner, Z3: An Efficient SMT Solver, Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, 2008, P. 1-10.
14. B. Dutertre and L. Moura. The YICES SMT Solver. In SMTCOMP: Satisfiability Modulo Theories Competition, 2006. http://yices.csl.sri.com/.
15. M. Barnett and W. Schulte. Contracts, Components, and their Runtime Verification. Technical Report MSR-TR-2002-38, Microsoft Research, April 2002. ftp://ftp.research.microsoft.com/pub/tr/tr-2002-38.pdf