МОВИ ТА МЕТОДИ СПЕЦИФІКАЦІЇ ПРОГРАМ

Методи формальної розробки, мови специфікацій

- Формальна специфікація – завершений опис моделі системи та вимог до її поведінки в термінах того чи іншого формального методу.
- Формальний метод (метод формальної розробки ПС) – це набір методів та інструментальних засобів, що базуються на математичних засадах (моделювання, математична логіка, теорія множин, теорія скінченних автоматів, алгебра, тощо), які використовуються для формальної специфікації, верифікації та аналізу вимог ПС та проблемних областей. Такі інструментальні засоби іноді ще називають системою формальних міркувань. Як правило, крім системи формальних міркувань формальні методи включають стандартизовані мови (мови специфікацій, формальні нотації). Приклади формальних методів: CSP, CCS, OBJ, VDM, RAISE, B-метод, Z-метод.

Методи формальної розробки, мови специфікацій

- Формальна мова (мова специфікацій, формальна нотація) – це мова з точно визначеним синтаксисом та семантикою. Мови формальних специфікацій використовуються для написання специфікації, тобто для опису властивостей певної проблемної області. Наприклад Z,B, CLEAR, ASL, Act One, LARCH.
- Формальна розробка – систематичне перетворення специфікації у виконуваний код з використанням певних формалізованих правил (певних формальних методів).

Методи формальної розробки, мови специфікацій

Методи формальної розробки базуються на використанні мови специфікацій та засобів формальних міркувань для побудови ПС. При цьому засоби формальних міркувань базуються на мовах формальних специфікацій, математичних засадах та використовуються для аналізу моделі та системи. В Мови формальних специфікацій в свою чергу базуються на математичному апараті (числення предикатів, алгебра, теорія скінченних автоматів) та використовуються для побудови моделі.

Існує низка підходів до методів формальної розробки:

- операційний підхід – розробка системи базується на описі моделі, яка має властивості розроблюваної системи;
- аксіоматичний підхід – розробка базується на аксіоматичному описі поведінки системи;
- алгебраїчний підхід – розробка базується на алгебраїчному описі властивостей дій;
- гібридний підхід об’єднує операційний підхід з аксіоматичним чи алгебраїчним підходами.

Кроки при формальній розробці об’єднані та супроводжуються аналізом, який базується на перевірці властивостей системи. Перевірка властивостей системи здійснюється засобами формальних міркувань за допомогою двох основних підходів:

- автоматичне доведення теорем (theorem proving) – програмне доведення логічних теорем (властивостей системи). В основі автоматичного доведення теорем лежить апарат математичної логіки.
- перевірка моделі (model checking) – процес перевірки, чи є побудована структура моделлю заданої проблемної області.

Класифікація методів та мов специфікацій програм

В основу класифікації мов та методів специфікацій програм можна покласти різні критерії: призначення, вид, повнота та форма представлення тощо. Таким чином отримуємо різні види класифікацій мов та методів специфікацій предметних областей та програмних систем.

У відповідності до повноти опису специфікації бувають виконуваними (Executable). Такі специфікації визначають поведінку системи достатньою мірою для забезпечення можливості їх запуску (анімації) на комп'ютері. Приклади мов специфікацій, які дозволяють задавати виконувані специфікації Z, RSL, VDM, B.

Класифікація методів та мов специфікацій програм

У відповідності до об’єкту специфікації виділяють наступні типи специфікацій:

- поведінкові (behavioral) специфікації – описують обмеження на поведінку об’єкту специфікації, а саме на функціональні можливості, безпеку та виконання;
- структурні (structural) специфікації – описують обмеження на внутрішній склад об’єкту специфікації, а саме на використання та склад, відношення залежності;
- специфікації взаємодії (interaction) – описують обмеження на взаємодію між двома чи більшою кількістю об’єктів специфікації, а саме на відповідність інтерфейсів та протоколів.

Класифікація методів та мов специфікацій програм

Методи специфікацій традиційно класифікують за підходом до представлення моделі. Зокрема, виділяють:

- моделе-орієнтовані підходи до специфікації. Метою таких специфікацій є побудова абстрактної моделі специфікованої інформаційної системи. Такі методи специфікацій базуються на описі станів (state-based). Приклади: VDM, Z, RAISE (RSL), B;
- методи орієнтовані на властивості (алгебричні). Метою таких специфікацій є опис системи в термінах бажаних властивостей без побудови явної моделі Такі методи специфікацій базуються на описі дій (action-based). Приклади: OBJ, Larch, Anna, Clear, StateCharts, CSP, CCS.

Класифікація методів та мов специфікацій програм

Відмінність між моделе-орієнтованими та алгебраїчними методами не настільки чіткі, як може спочатку здатися. На практиці моделе-орієнтовані специфікації часто описують аспекти специфікуємої системи за допомогою аксіом, що властиве специфікаціям орієнтованим на властивості. А алгебраїчні специфікації часто описують набори базових типів даних при моделюванні властивостей, але при цьому використовують побудову моделі специфікуємої системи. Методи, які є одночасно моделе-орієнтованими та орієнтованими на властивості називаються гібридними. Зокрема, VDM, Anna, Z, RSL, UML.

Класифікація методів та мов специфікацій програм

При класифікації мов специфікацій за призначенням виділяють універсальні мови специфікацій, які можуть бути застосовані для опису широкого класу задач (Z, B, VDM, Clear, RSL) та спеціалізовані мови специфікацій, що розроблені для певних проблемних областей та є практично незастосовними для опису властивостей інших проблемних областей. Так, CCS та CSP є спеціалізованими методами формальних специфікацій, що призначені для опису взаємодії між паралельно працюючими процесами.

Можна класифікувати дескриптивні системи (мови) за видом представлення специфікації. З цієї точки зору можна виділити візуальні (графічні) та текстові специфікації. У візуальних специфікаціях поведінка та структура системи представляється графічним способом (у вигляді графів). Такими є UML/OCL, StateCharts.

Класифікація методів та мов специфікацій програм

Крім того, мови специфікацій класифікують у відповідності до типу представлення, при цьому мови специфікацій розподіляються на наступні базові типи:

- транзиційні специфікації (специфікації переходів станів);
- темпоральні (часові) логічні специфікації;
- паралельні специфікації;
- специфікації абстрактної моделі;
- алгебричні специфікації;
- аксіоматичні специфікації.

Одна мова специфікацій може підтримувати кілька типів.

Класифікація методів та мов специфікацій програм

Специфікації абстрактної моделі

Явно описують поведінку в термінах моделі. Специфікація використовує чітко визначені типи (множини, послідовності, відношення, функції) та визначає операції на моделях.

Специфікація включає модель типу, інваріант, перед- та пост-умови.

Особливостями таких специфікацій є явне моделювання станів в моделях, можливість виконання, можливість побудови об’єктів в ієрархічному порядку.

Специфікації абстрактної моделі підтримуються такими мовами VDM, Z, RAISE (RSL), UML/OCL.

Класифікація методів та мов специфікацій програм

Алгебричні специфікації

Визначають поведінку сукупністю відношень еквівалентності, що описують властивості об’єктів та операції над ними.

Специфікація включає функції та відношення.

Особливостями таких специфікацій є наявність описів функцій, підтримка абстракції даних, особливо застосовні для абстрактних типів даних.

Алгебраїчні специфікації підтримують такі мови, як OBJ, Larch, Anna, Clear.

Класифікація методів та мов специфікацій програм

Транзиційні системи

Описують поведінку системи набором станів, та визначають операції як переходи між станами чи відношеннями між станами.

Специфікація включає стани та переходи.

Особливостями таких специфікацій є загальний простір станів, текстові та графічні нотатки, модульність, застосування до систем керування.

Специфікації станів та переходів між ними підтримують такі мови, як StateCharts, CSP, CCS, UML/OCL (діаграми скінченних автоматів (станів), діяльності).

Класифікація методів та мов специфікацій програм

Аксіоматичні специфікації

Визначають поведінку за допомогою логічних формул, а також задають вхідні, проміжні та вихідні твердження.

Специфікація включає операції зв'язку між вхідними та вихідними параметрами, аксіоми з перед- та пост- умовами.

Особливостями аксіоматичних специфікацій є:

- широкі можливості застосування;
- можливість розширення системи;
- використання методів доведення.

Аксіоматичні специфікації підтримують такі мови, як VDM, Anna та Z.

Класифікація методів та мов специфікацій програм

Темпоральні та паралельні специфікації

Явно визначають поведінку у відповідності з описами системних станів та наборів подій в термінах порядку виконання та синхронізації. Паралельні специфікації визначають дії в термінах паралельних подій.

Специфікація включає стани, переходи, події, впорядкування та синхронізацію.

Особливостями таких специфікацій є потужний механізм специфікації, рівень синхронізації специфікації.

Часові логічні специфікації підтримують такі мови, Temporal Logic Specification (TLS), Petri nets, TLS, StateCharts, GIL, CSP, UML/OCL (діаграми скінченних автоматів (станів), діяльності, комунікації, взаємодії, послідовності, синхронізації).

Особливості різних методів специфікацій програм

Деякі особливості різних мов специфікацій проілюструємо на прикладі адресної книги. Адресна книга зберігає записи про осіб та їхні адреси. При цьому одна особа може мати лише одну адресу, але за однією адресою може проживати кілька осіб. Специфікуєма ПС повинна забезпечувати можливість додавання та видалення інформації про особу та її адресу, а також знаходження адреси за іменем особи.

Специфікації абстрактної моделі. Аксіоматичні специфікації (Z)

Нехай максимальна кількість елементів в адресній книзі – *MaxSize*.

$$MaxSize : \mathbb{N}$$
$$MaxSize \leq 65535$$

Можливі повідомлення:

MESSAGE ::= *success* | *found* | *overflow* | *known* | *unknown*

Стан системи, описаний в Z, виглядає наступним чином:

AddressBook
$$address : PERSON \pfun ADDRESS$$
$$person : \mathbb{P}\ PERSON$$
$$person = \mathrm{dom}\ address$$
$$\#person \leq MaxSize$$

Специфікації абстрактної моделі. Аксіоматичні специфікації (Z)

Тут і далі стани представлені як множина імен осіб *person* та часткова функція *address*, яка зіставляє особі її адресу.

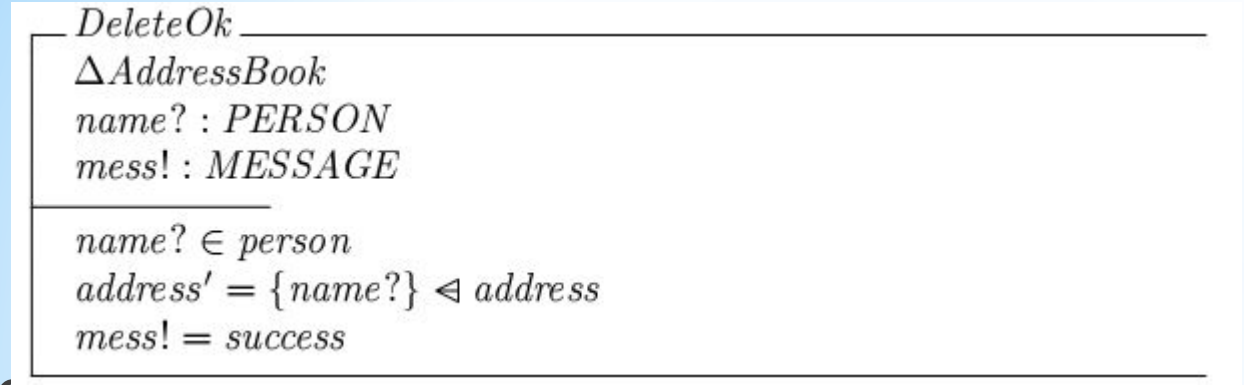
Операція додавання нового запису може бути специфікована наступним чином:

InsertOk

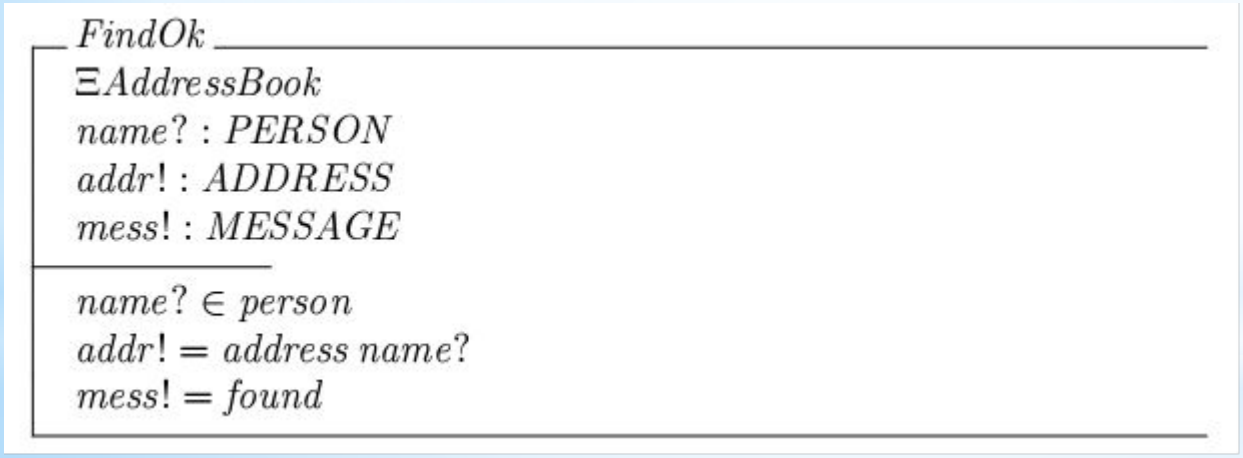
$$\Delta AddressBook$$
$$name? : PERSON$$
$$addr? : ADDRESS$$
$$mess! : MESSAGE$$

$$name? \notin person$$
$$\#person < MaxSize$$
$$address' = address \cup \{name? \mapsto addr?\}$$
$$mess! = success$$

Операції видалення запису може бути специфікована наступним чином:



Операція пошуку адреси за іменем може бути специфікована наступним чином:



Компоненти станів, які відносяться до операції, разом з їх типами, зазначаються після оголошення схеми даних Δ *AddressBook*, так у випадку операції *InsertOk* маємо два вхідні параметри *name*? та *address*?, а у випадку операції *FindOk* маємо один вхідний параметр *name*?, та один вихідний (шуканий) – *address*!. Вхідні параметри позначаються знаком “?” після імені параметру, а вихідні – “!”. В пост-умові нові значення станів компонент визначені їхнім іменем зі штрихом (напр. $address'$). У випадку операції *FindOk* компонента стану системи *address* незмінна, призначена лише для читання, тобто її заключне значення рівне початковому ($address' = address$), це явно в схемі не зазначено, але позначення Ξ перед іменем схеми даних *AddressBook* означає незмінність даних цієї схеми.

Опишемо ініціалізацію, тобто стан, який виникає перед першим використанням бази даних. При цьому адресна книга немає жодного запису.

DataBookInit

AddressBook

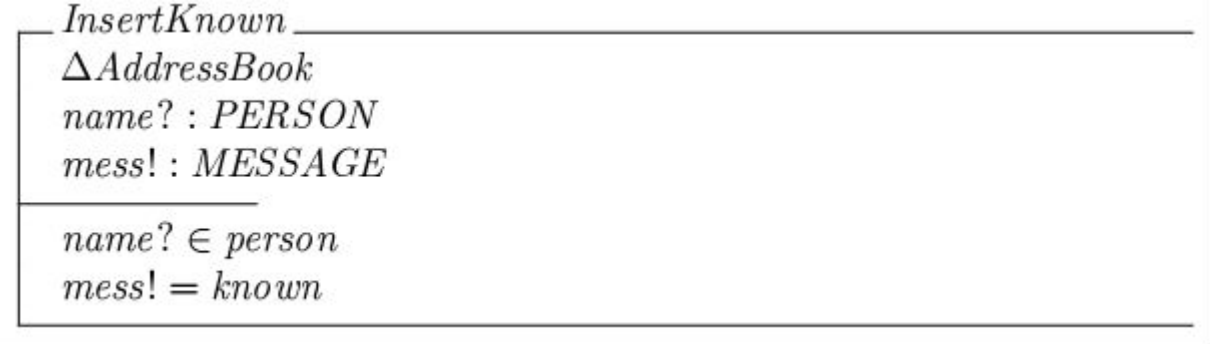
address = ∅

Специфікація описує поки що просту базу даних “Адресна книга”, але некоректні вхідні дані можуть привести до помилок в описаних операціях. Числення схеми може використати додаткове розширення специфікації для того, щоб вказати на зроблені при вводі даних помилки.

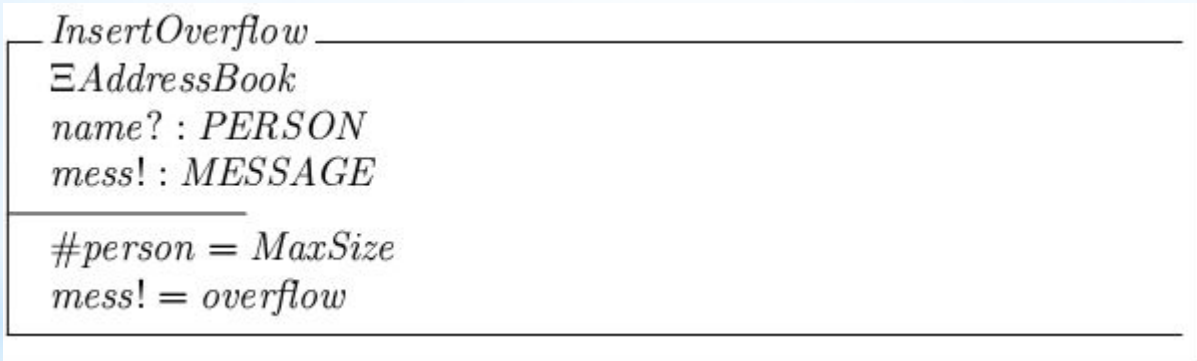
Почнемо з опису множини нових подій для бази даних. Передумова кожного випадку описує обставини, при який дія може зазнавати невдачі, та пост-умова визначає, що стан системи залишається незмінним.

Кожен з випадків помилки вводу матиме один вихідний параметр – *message*! для повідомлення про помилку. Параметр *message*! є рядком.

При виконанні операції *InsertAddress* виникне помилка, якщо *name*? ∈ *person*, тобто адреса для цієї особи вже відома, тому маємо:



Також, при виконанні операції *InsertAddress* може виникнути помилка переповнення, тому маємо:



Тепер можна описати операцію *Insert*:

Insert ≙ *InsertOk* ∨ *InsertOverflow* ∨ *InsertKnown*.

Аналогічно, при виконанні операції *FindOk* виникне помилка, якщо *name*? ∉ *person*, тобто адреса для цієї особи невідома, тому маємо:

Тоді операція *Find* матиме вигляд:

Find≙ *FindOk* ∨ *FindUnknown*.

Аналогічна помилка може виникнути при виконанні операції *DeleteOk* тому маємо:

Delete ≙ *DeleteOk* ∨ *DeleteUnknown*.

Нові версії операцій були визначені, як складні операції, що є диз'юнкцією схем, відповідних за нормальне виконання операції та обробки помилок.

Слід зауважити, що існує можливість трансляції деяких UML-діаграм в Z-специфікації.

Двома основними особливостями стилю Z специфікацій є наступні:

схеми використовуються для опису всіх аспектів системи при обговоренні: стани, які вона може приймати, можливі переходи від одного стану до іншого;

кілька схем можуть бути об’єднані для побудови опису складних об’єктів.

Метод VDM. Мова VDM-SL

Метод VDM (Vienna Development Method) розроблений Віденською лабораторією компанії IBM та розвивався далі в роботах Д. Бьорнера (Dines Bjorner) та К. Джонса (Cliff Jones). В своїх цілях VDM та Z досить схожі, але між ними є ціла низка відмінностей. Кожен з цих методів має свої переваги та недоліки.

Розглянемо специфікації на мові VDM для запропонованого прикладу. Стан системи, описаний в VDM виглядає наступним чином:

AddressBook::

person: **set of** PERSON

address: **map** PERSON **to** ADDRESS

where

inv_ *AddressBook* ≙ *person* = **dom** *address*

Як і в Z компоненти стану задані типами та інваріантними відношеннями між цими типами.

Стиль VDM уникає надлишковості компонент станів, тому компонента *person* була б, напевно, виключена, так як вона може бути отримана з іншої компоненти *address* за інваріантом.

Операції додавання нового запису та знаходження адреси особи можуть бути специфіковані наступним чином:

AddAddress(*name* : PERSON, *address* : ADDRESS)
ext wr *person* : **set of** PERSON
wr *address* : **map** PERSON **to** ADDRESS
pre
$name \notin person$
post
$person' = person \cup name \ \wedge$
$address' = address \cup \{name \mapsto address\}$.
FindAddress(*name* : PERSON) *address* : ADDRESS
ext wr *person* : **set of** PERSON
rd *address* : **map** PERSON **to** ADDRESS
pre
$name \in person$
post
address =*address*(*name*)

Компоненти станів, які відносяться до операції, разом з їх типами, внесені до списку **ext** в пункті, який відмічено як перезаписуємий **wr**. В **post**-умові нові значення станів компонент визначені їхнім іменем зі штрихом (напр. *address*′). Якщо деякі компоненти станів операції призначені лише для читання, тобто їх заключні значення рівні початковим, то вони мають бути занесені до списку **ext** в пункті, який відмічено як призначений лише для читання **rd**, а тому факт, що їх значення незмінне не потрібно явно зазначати в **post**-умові. Вхідні та вихідні параметри операції описуються в заголовку операції наступним чином: *Назва-операції*(*Вхідний параметр_1*: Тип_вх_1, …, *Вхідний параметр_n*: Тип_вх_n) *Вихідний параметр_1*: Тип_вих_1, …, *Вихідний параметр_m*: Тип_вих_m.

Інваріант на заключному стані не є неявною частиною **post**-умови, як у Z, тому необхідно визначити заключні значення отриманих станів компонент таких як *address*. Це може зробити VDM специфікації менш короткими ніж відповідні Z специфікації, але це є корисною надмірністю в специфікації. Оскільки **pre-** і **post-**умови відділені, можна формулювати доведення при написанні специфікації: для специфікації, щоб бути правильно побудованою, **pre-**попередня умова повинна гарантувати існування заключного стану, який задовольняє **post**-умову, і всі такі заключні стани повинні задовольняти інваріанту.

Хоч однаково можливо записати **pre**-умову окремо в Z і доводити теорему про правильну побудову, стиль VDM забезпечує підтримку для цього, вимагаючи щоб заключні значення станів усіх компонентів були визначеними. Наявність окремої **pre-**умови також робить більш легким формулювання правил для доведення правильності виконання.

VDM не має аналогу для числення схеми в Z, яке дозволяє утворювати більші специфікації з менших. Можливо вказати **pre**- і **post**-умови однієї дії в описі іншої, але специфікації дії не розглядаються як окремі об'єкти, які можуть бути об'єднані для створення нових.

Багато подібного між Z і VDM. Вони обидва використовують звичайні математичні структури – множини, функції і послідовності – як моделі даних, а також нотації, предикати логіки для опису дії на даних. В обох методах специфікація складається з опису набору станів, що супроводжується описом дій, які змінюють стан.

Алгебраїчні методи. Мова Clear

Основним словником мов алгебраїчних специфікацій є лише деякі іменні множини та тотальні функції на цих множинах. Специфікації описують властивості, яким ці функції повинні задовольняти, типово, задаючи рівняння, що зв'язують функції між собою.

Для прикладу розглянемо базу даних "Адресна книга" на алгебраїчній мові специфікацій Clear. Маємо два основних атрибути запису бази даних: імена та адреси. Ми повинні мати можливість повідомити, якщо вхідне ім'я співпадає з вже задіяним іменем. Для цього розширимо *Bool* типу boolean новою іменною множиною, бінарною операцією == порівняння пари імен.

const *Name* =

enrich *Bool* **by**

sorts *person*

opns _==_: *name*, *name* → *bool*

eqns *n* == *n* = *true*

n == *m* = *m* == *n*

n == *m* ∧ *m* == *p* ⇒ *n* == *p* = *true*

enden

При заданні операції порівняння пари імен задана її сигнатура: **opns** _==_: *name*, *name* → *bool*, а також рефлексивність, сиетричність та транзитивність цієї операції *n* == *n* =true, *n* == *m* = *m* == *n*, *n* == *m* ∧ *m* == *p* ⇒ *n* == *p* = *true*.

Для адрес ніяка перевірка на рівність не потрібна, але повинна бути спеціальне дане *unknown* типу address, яке є результатом спроби знайти адресу за іменем, якого немає в базі даних.

```
const Address =
    theory
        sorts address
            opns unknown: address
    endth
```

Елементи помилок, подібно до невідомого об’єкту є необхідними в алгебраїчних специфікаціях у відповідності до вимоги, що всі дії мають бути повними, тобто коли дія застосовується поза її областю дії, то результатом є елемент помилки. Мова специфікацій передбачає наявність спеціальних засобів для представлення елементів помилки.

Можливі стани бази даних описані в термінах шляху, яким вони можуть бути створені. Початковий стан бази даних *empty*, тобто це стан, що не містить жодної адреси. Новий стан може бути отриманий зі старого, додаванням нової особи та нової адреси операцією *AddAddress*. Ця операція має три аргументи ім'я, адресу, базу даних та повертає результат додавання імені і адреси до бази даних. Сигнатура цієї операції визначена наступним чином: *AddAddress* : *name*, *address*, *db* → *db*.

Параметр *DataBase* вимагає збагачення дій *AddAddress* та *empty*. Дія *empty* константна по відношенню до параметру *db*, тобто функція, яка не має жодних аргументів.

const *DataBase* =

enrich *Name* + *Address* **by**

data sorts *db*

opns *empty*: *db*

AddAddress : *name*, *address*, *db* → *db*

eqns *AddAddress*(*n*, *x*, *AddAddress*(*n*, *y*, *d*))

= *AddAddress*(*n*, *y*, *d*)

AddAddress(*n*, *x*, *AddAddress*(*m*, *y*, *d*))

= *AddAddress*(*m*, *y*, *AddAddress*(*n*, *x*, *d*))

if *not*(*n* == *m*)

enden

Не існує ніяких елементів *db*, крім тих, які можуть бути вище визначені, починаючи з порожньої бази додаванням нових імен та адрес. Задані властивості додавання адрес. Якщо дві адреси добавлені для одного і того ж імені, то остання адреса заміняє першу (*AddAddress*(*n*, *x*, *AddAddress*(*n*, *y*, *d*)) = *AddAddress*(*n*, *y*, *d*)), якщо імена відмінні, то порядок додавання адрес не має значення (*AddAddress*(*n*, *x*, *AddAddress*(*m*, *y*, *d*)) = *AddAddress*(*m*, *y*, *AddAddress*(*n*, *x*, *d*)) **if** *not*(*n* == *m*)). Таким чином, кожен вираз бази даних може бути приведений до канонічної форми, в якій кожне ім’я добавлено щонайбільше один раз. Два вирази в такій формі рівні, якщо вони відрізняються лише за порядком додавання імен.

Цей опис бази даних демонструє відмінність між специфікаціями, заданими в специфікаціях абстрактної моделі Z чи VDM та заданим в алгебраїчній мові Clear. В Z та VDM явна модель для станів давалася в термінах множин та функцій, а в Clear стани описані неявно як ті об’єкти, що можуть бути отримані застосуванням деяких визначених рівняннями дій.

Специфікація системи бази даних завершена, додамо операцію *FindAddress* для знаходження адреси. Аргументами цієї операції є ім'я та база даних, а повертає ця операція адресу, пов'язану з іменем в базі даних, якщо таке ім'я є в базі даних, та невідоме значення адреси інакше. Таким чином операція має сигнатуру **opns** *FindAddress* : *name*, *db* → *address*.

const *AddressBook* =

enrich *DataBase* **by**

opns *FindAddress* : *name*, *db* → *address*

eqns *FindAddress* (*n*, *AddAddress*(*n*, *x*, *d*)) = *x*

FindAddress (*n*, *AddAddress*(*m*, *x*, *d*))

= *FindAddress* (*n*, *d*) **if** *not* (*n* == *m*)

FindAddress (*n*, *empty*) = *unknown*

enden

Clear забезпечує засоби для процедур, які можуть містити базові припущення як формальні параметри. Наприклад, специфікація бази даних могла б бути побудована як процедура з наступним заголовком:

proc *AddressBook* (*Name*: *Ident*, *Address* : *Triv*) =

…

Типами параметрів *Name* та *Address* є *Ident* та *Triv* відповідно. *Name* повинно мати перевірку на рівність ==. Якщо специфікація задана в такій формі, то її можна застосовувати до різних фактичних параметрів, щоб визначити різні версії бази даних. Наприклад, база даних, в якій імена та адреси є рядками, визначається наступним виразом:

AddressBook (String[element **is** *string], String*[*element* **is** *string*]).

Така конструкція особливо корисна для визначення основних типів даних типу множин та послідовностей. Можна говорити про послідовності символів, послідовності натуральних чисел, множини послідовностей натуральних чисел, і таке інше. Невелика незручність – те, що кілька типів можуть тоді мати одну і ту ж саму назву.

Ці засоби роблять можливим побудову потужних колекцій з параметризованих теорій, що містять всі звичайні типи. Поняття метавиду дозволяє визначати складні вимоги до параметрів теорії: наприклад, можна будувати теорію сортування, в якій параметр описує не лише тип об’єктів, які потрібно сортувати, але також і відношення порядку, яке потрібно використовувати для сортування.

Існують також роботи з алгебраїчних специфікацій часткових функцій.

Специфікації абстрактної моделі. Метод RAISE. Мова RSL

Проект RAISE був розроблений компаніями: Dansk Datamatik Center (DDC), Computer Resources International (CRI), STC Technology Ltd (STL) (тепер входить до складу BNR Europe Limited), Nordisk Brown Boveri (тепер SYPRO), ICL, ESPRIT. До складу RAISE входить мова RSL, а також велика кількість методів і засобів для виконання формального розвитку і доведень властивостей ПС.

RAISE ґрунтується на ідеях, відображених у багатьох формальних методах і мовах специфікацій. Для розробки цього проекту компанії DDC/CRI і STL у 1983 виконали формальну оцінку методів [65, 66], було розглянуло велику кількість підходів з метою виділити можливу основу для проекту RAISE. Вони прийшли до висновку, що модельно-орієнтовані підходи до специфікації програм, такі як VDM, є найбільш життєздатними для індустріального використання. Однак метамови VDM було недостатньо, тому що вона не працювала з структуруванням і паралелізмом, що мало місце в алгебраїчних мовах специфікацій. Також алгебраїчні мови дозволяють задавати різні рівні абстракції. Так, наприклад, модельно-орієнтовані мови оперують конкретними типами, у той час як алгебраїчні мови – абстрактними типами. Таким чином, задача, що постала перед проектувальниками RSL, полягала в тому, щоб побудувати структуру, в якій основні особливості VDM могли бути розширені засобами алгебраїчних мов специфікацій для досягнення структурування і паралелізму.

Існує багато різних версій метамови VDM. На ранній стадії розробки проекту RAISE була зроблена спроба об'єднання різних версій VDM. Деякі узгоджені в різних версіях VDM аспекти були включені як у ранні, так і в заключні версії RSL: типи конструкторів для відображень (Maps), множин (Sets), списків (Lists), декартових добутків (×) і т.д., разом із застосовуваними формами виразів і визначень функції (включаючи **pre-**/**post-**стиль). Типи в RSL повинні бути явно визначені (як в VDM).

У RSL є два види модулів: об'єкти і схеми. Схема представляє (можливо параметричний) клас моделей, у той час як об'єкт представляє єдину модель, що належить до зазначеного класу або множини таких моделей. Схеми RSL беруть об'єкти як параметри і повертають в якості результатів класи.

Розглянемо варіант представлення схем RSL на прикладі бази даних “Адресна книга”:

Специфікація, орієнтована на задання абстрактних операцій:

scheme
ABS_ADDRESS_BOOK =
class
type
Person, Address, Database
value
empty : Database,
hasPerson : Person × Database → **Bool**,
deleteAddress : Person × Database → Database,
addAddress : Person ×Address×Database −~→ Database,
findAddress : Person × Database −~→ Address
axiom
∀p∈Person·hasPerson(p, empty) ≡ **false,**
∀p∈Person· ∀a∈Address· ∀db∈Database ·
hasPerson(p, addAddress((p, a), db)) ≡ **true,**
∀p∈Person· ∀db∈Database·hasPerson (p, deleteAddress(p, db)) ≡ **false,**
∀p∈Person· ∀db∈Database· (hasPerson(p, db) = **false**)
⇒ deleteAddress(p, db) ≡ db
∀p∈Person· ∀db∈Database· deleteAddress(p, db) ≡
deleteAddress(p, deleteAddress(p, db),
∀p∈Person· ∀a, a_1 ∈Address·
∀db∈Database·addAddress(p, a, addAddress(p, a_1, db)) ≡ addAddress(p, a, db),
∀p, p_1∈Person· ∀a∈Address· ∀db∈Database·
findAddress(p, addAddress(p_1, a, db)) ≡
if (p≠p_1) **then** findAddress(p, db) **else** a **end**
end

Специфікація, орієнтована на множини:

scheme
SET_ADDRESS_BOOK =
class
type
Person = Text
Address = Text
Address_book = Person × Address
Database = Address_book-**set**
value
empty : Database ≡ {},
hasPerson : Person × Database → **Bool**
hasPerson(p, db)≡(∃ a : Address · (p, a)∈db),
findAddress : Person × Database −~→ Address
findAddress(p, db)≡a : Address·(p, a)∈db **pre**
hasPerson(p, db),
deleteAddress : Person × Database → Database
deleteAddress(p, db)≡(db\{(p, findAddress(p, db))}) **pre**
hasPerson(p, db),
addAddress : Address_book × Database −~→ Database
addAddress((p, a), db)≡
if hasPerson(p, db) **then** addAddress((p, a),
deleteAddress(p, db)) **else** (db∪{(p, a)}) **end**
end

Специфікація, орієнтована на відображення:

```
scheme
MAP_ADDRESS_BOOK =
    class
        type
Person = Text
Address = Text
            Address_book = Person → Address
Database = Address_book-set
            value
                empty : Database ≡ [],
hasPerson : Person × Database → Bool
hasPerson(p, db) ≡ p∈dom db,

                findAddress : Person × Database −~→ Address
                findAddress(p, db) ≡ db(p) pre hasPerson(p, db)
        deleteAddress : Person × Database → Database
(db \ {p}) pre hasPerson(p, db),
addAddress : Address_book × Database −~→ Database
                addAddress((p, a), db) ≡ (db†[p↦a])
        end
```

Можна довести, що SET_ADDRESS_BOOK є коректною конкретизацією ABS_ADDRESS_BOOK, а також, що MAP_ADDRESS_BOOK є коректною конкретизацією ABS_ADDRESS_BOOK, для цього потрібно перевірити коректність аксіом.

Темпоральні (часові) та паралельні специфікації та транзиційні специфікації

Діаграма станів UML (Statechart diagram)

Діаграма станів базується на Statechart. Діаграма станів визначає усі можливі стани, в яких може знаходитися конкретний об’єкт, а також процес зміни станів об’єкту в результаті деяких подій.

Діаграма станів для БД «Адресна книга».

Рівні застосування формальних методів (ФМ)

Визначивши, що ФМ вам дійсно потрібні, вибравши придатну нотацію і зрозумівши, які компоненти системи виграють від формального підходу, необхідно подумати про рівень, до якого будуть застосовуватися ФМ. Будемо вважати, що таких рівнів три:

- формальна специфікація;
- формальна розробка та перевірка;
- доведення.

Формальна специфікація

Використання формальної специфікації вигідно в багатьох випадках. Формальна мова допомагає зробити специфікації більш повними й однозначними, полегшує їхнє обмірковування, навіть на неформальному рівні. Він дозволяє краще орієнтуватися в рівнях абстрагування і відкладати складності до більш придатного моменту.

ФМ особливо важливі для проникнення в суть проектованої системи, дозволу неоднозначностей і структурування як підходу до проблеми, так і реалізації, що виходить у результаті.

Ця техніка дуже добре зарекомендувала себе при розробці програмної архітектури для сімейства осциллографів і в таких різноманітних видах діяльності, як формальний опис алгоритму для системи голосування, опис структури документів і виявлення внутрішніх протиріч у побудові WWW.

Формальна розробка і перевірка

Повна формальна розробка в даний час проводиться досить рідко. Вона передбачає формальну специфікацію системи доведення, що система має необхідні властивості і не має небажаними, і нарешті, застосування уточнюючих розрахунків, що перетворюють абстрактну специфікацію в усі більш і більш конкретні представлення, останнім з який є код, що виконується.

У цьому випадку допускаються як формальні, так і неформальні, але строгі доведення.

Доведення

З появою інструментів підтримки, зокрема, програм для доведення теорем і перевірки доведень, стала можлива механічна перевірка несуперечності й обґрунтованості доведень.

Для деяких класів систем машинна перевірка доведень дійсно дуже важлива. Природно включити сюди системи з підвищеними вимогами до безпеки і захисту. Фактично за таку перевірку ратують у своїх стандартах багато організацій. Так, Європейське космічне агентство наполягає на застосуванні формальних виведень (перед тестуванням) скрізь, де це можливо, і рекомендує незалежну перевірку доведень як спосіб зменшити імовірність помилки людини.

Деякі ФМ включають системи виведення теорем у якості однієї з компонентів. Це в першу чергу HOL, Larch (з компонентом LP - Larch Prover), OBJ і PVS. Крім того, існують системи доведення і середовища підтримки, що включають подібні системи, для таких методів як B (B toolkit фірми B-Core), CSP (FDR фірми Formal Systems (Europe) Ltd.), RAISE (CRI), VDM (VDM Toolbox фірми IFAD) і Z (Balzac/Zola фірми Imperial Software Technology, ProofPower фірми ICL).

Цікава ідея розробки систем доведення для використання з визначеним методом. Так, програми доведення теорем для Z були створені в середовищах EVES, HOL і OBJ.

Кожен із трьох перерахованих рівнів корисний сам по собі. Але перед тим, як стати на шлях цілком формальної розробки з машинною перевіркою доведень, варто оцінити, чи окупляться додаткові витрати часу, сил, людської праці, грошей на інструментальні програми і т.д. Для систем, що вимагають найвищої надійності – таких, де помилка загрожує втратою людських життів, значним фінансовим чи матеріальним збитком – вони окупаються і, більш того, необхідні.