МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2008
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2008
Proceedings of the Thirty Seventh Spring Conference of
the Union of Bulgarian Mathematicians
Borovetz, April 2–6, 2008

ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ НА РЕАЛИЗАЦИИТЕ НА АБСТРАКТНИ ТИПОВЕ ДАННИ*

Магдалина Василева Тодорова

Статията е посветена на абстрактните типове данни в университетското обучение по програмиране. Съществено внимание е отделено на верификацията и валидацията на реализациите на абстрактните типове данни.

1. Увод. Изучаването на абстрактните типове данни (АТД) е обект както на училищния, така и на университетския курс за обучение по програмиране. Чрез АТД се разработват абстрактни модели за данни и начини за обработка на тези данни, което е важен компонент в процеса на решаване на задачи чрез компютър. Знанията за АТД са полезни за разбирането и практиката на обектно-ориентирания анализ, за проектирането и за програмирането.

В статията е описан подход за създаване на АТД, използван в обучението по обектно-ориентирано програмиране във Факултета по математика и информатика на СУ "Св. Кл. Охридски". Освен традиционните фази проектиране и реализиране [1, 2], подходът включва и фазите верификация и/или валидация на реализациите на АТД.

Важна част на процеса на програмиране е проверката дали създадените програми са правилни. Основни методи за търсене на грешки са имитационното моделиране, тестването, дедуктивният анализ и верификацията [4]. За целите на обучението най-често се използва тестване, при което за допустими входни данни се проверява дали се получава правилен резултат. Но дори и след завършване на тестването не може да сме съвсем сигурни, че програмата е правилна.

В редица технологии, като управление на транспорта, електронен бизнес, медицински системи, телефонни мрежи и др., грешка в програма може да струва много скъпо. Например, неправилен превод на число от 64-тична в 16-тична система доведе до взривяването на ракетата *Ariane* 5 и причини загуби за над 370 млн. долара [6, 7]. Също поради елементарна грешка, медицинският ускорител *Therac* 25 причини 6 смъртни случая [6]. Затова за реализиране на подобни технологии е желателно да се използват формални подходи за доказване, че програмата е правилна. Такива подходи са методите за верификация на програми.

2. Проектиране и реализиране на АТД. Абстрактният тип от данни е описание на множество от стойности (данни) и множество от операции, които могат да

*Статията описва резултати по договор 83/2007 "Принципи, методи и средства на компютърното моделиране (теория, практика, обучение)" на ФНИ към СУ "Св. Кл. Охридски".

бъдат изпълнявани над данните [1]. Абстрактността на типа от данни е в смисъл, че типът от данни не зависи от различните негови реализации. Проектирането на АТД включва описания на данните и на операциите. Описанието на данните определя типовете на компонентите на данните и връзките между тях, както и множеството от допустими стойности. Описанието на операциите включва: синтаксис, семантика, пред- и пост-условия. Фиг. 1 дава пример за декларация на АТД Queue, представяща проекта на структурата от данни опашка от цели числа. Декларацията се състои от три части: *данни*, *операции* и *спецификации*.

```
АТД Queue
{ данни
    int a[front..rear);                              // елементи на опашката
    int front,                                         // начало на опашката
       rear;                                             // край на опашката
 операции
    void Initialization();                          // създава празна опашка
    bool Empty();                             // проверява дали опашка е празна
    void Push(int x);                               // включва елемент в опашка
    int Pop();                          // изключва елемент от непразна опашка
    int Length();                       // намира броя на елементите на опашка
 спецификации
{ true } Initialization() { a[front..rear) = [] }                          (1)
{ front ≤ rear } empty() { return = (front = rear) }                       (2)
{ front ≤ rear } A = a; Front = front; { (a[front..rear) =
                   A[Front..Rear) append [x] )                             (3)
           Rear = rear; Push(x)^(front = Front)^(rear = Rear+1)}
{ front < rear } A = a; Front = front; { (return = a[Front])^              (4)
           Rear = rear; Pop()  (a[front..rear) = A[Front+1..Rear))^
                      (front = Front+1) ^ (rear = Rear) }
{ front ≤ rear } A = a; Front = front; { (return = rear - front) ^         (5)
           Rear = rear; Lenth() (a[front..rear) = A[Front..Rear)) ^
                      (front = Front)^(rear = Rear) }
};
```

Фиг. 1. Пример за декларация АТД Queue

Частта *данни* представя опашката като редица от цели числа $\mathtt{a}_{\mathtt{front}}$, $\mathtt{a}_{\mathtt{front+1},\ldots}$, $\mathtt{a}_{\mathtt{rear-1}}$, където `front` ≤ `rear`. Условието `front` ≤ `rear`, определя съществуването на редицата и се нарича *инварианта на данните* на АТД Queue. Съкратено редицата е записана чрез `a[front..rear)`. Началото на опашката е означено с `front`, а краят ѝ – с `rear` (`rear` е указател след последния елемент на опашката). Ако `front = rear`, опашката е празна (означава се с [ ]).

В часта *операции* са декларирани операциите: създаване на празна опашка, проверка дали опашката е празна, включване и изключване на елемент в/от опашка и намиране на броя на елементите на опашка. Тези декларации задават синтаксиса на операциите. Семантиката им, както и пред- и пост-условията са зададени в частта *спецификации* (съкратено от функционални спецификации). За описанието на спе-

цификациите са използвани тройките на Хоар {Q}S{R}, където Q и R са предикати, а S – програма или програмен фрагмент. Означението {Q}S{R} има следната интерпретация: ако изпълнението на S започне от състояние, удовлетворяващо Q, изпълнението на S завършва в състояние, удовлетворяващо R [5]. Чрез return в предикатите R на спецификациите е означена върнатата след изпълнението на съответната операция стойност, а чрез append – конкатенацията на две редици (a[n..m+1) append b[k..p+1) е редицата a[n], a[n+1], ..., a[m], b[k], b[k+1], ..., b[p]). За записване на предикатите от спецификациите са използвани математически означения, а за програмните фрагменти – езикът C++. Така спецификацията (1) се интерпретира по следния начин: ако изпълнението на Initialization() започне при начално състояние true, то винаги завършва и конструираната опашка е празната. Програмният фрагмент A = a; Front = front; Rear = rear; Push(x) в спецификация (3) запомня в A, Front и Rear данните, представящи текущата опашка, след което прилага Push(x) и получава нова опашка с данни a, front и rear. (3) се интерпретира така: ако изпълнението на A = a; Front = front; Rear = rear; Push(x) започне от състояние, при което е в сила front $\leq$ rear (т.е. опашката е коректно зададена), то завършва в състояние, при което редицата, представяща новата опашка a[front..rear) е равна на конкатенацията на редицата, представяща старата опашка A[Front..Real) с редицата, състояща се от елемента x; началото на новата опашка съвпада с началото на предишната, а краят на новата опашка е по-голям с 1 от края на предишната.

Декларацията на АТД, илюстрирана по-горе, в частите данни и операции е аналогична на декларацията на структура в езика C++ [3, 5]. За да се използва, се дефинира конкретна опашка q по следния начин:

Queue q;

Аналогично на ООП, с опашката q са свързани: q.a – носител на q, q.front и q.rear – указатели към началото и след края на опашката q, q.Initialization() – инициализира q с празната опашка, q.Empty() – проверява дали q е празна опашка, q.Push(x) – включва елемента x в края на опашката q, q.Pop() – връща елемента от върха на q, след което го изключва от q. Последните разсъждения показват, че всяка от декларираните операции има неявен параметър от абстрактния тип от данни Queue. Например операцията Push реално има следната декларация: void Push(Queue q, int x).

Могат да се направят множество реализации на АТД опашка. На фиг. 2 е дадена една опростена реализация. Причината за това е, да бъде използвана за илюстриране на процеса на верификация. Използван е езикът за ООП C++. Дефинициите на операциите са вградени в декларацията на структурата Queue и трябва да удовлетворяват съответните спецификации от декларацията на АТД Queue.

Реализацията на АТД е коректна (тотално коректна, правилна), ако всяка нейна операция е реализирана коректно (удовлетворява съответната си функционална спецификация) и запазва инвариантата на данните на типа.

3. Верификация и валидация на реализация на АТД. Да се верифицира реализация на АТД означава да се верифицира реализацията на всяка операция и да се докаже, че всяка операция запазва инвариантата на данните на АТД.

В случая на АТД Queue, за верификация на операциите му трябва да се докаже верността на всяка спецификация от фиг. 1, като се използват реализациите на

```
struct Queue
{ int a[size];                          // контейнер на елементите на опашката
   int front, rear;      // указатели към началото и след края на опашката
   void Initialization()
     { front = 0; rear = 0;
     }
   bool Empty()
   { return front == rear;
   }
   void Push(int x)
   { a[rear] = x; rear++;
   }
   int Pop()
   { int x;
     if (rear - front > 0)
       { x = a[front]; front = front + 1;
       }
     else x = −1;
     return x;
   }
   int Length()
   { return rear - front;
   }
};
```

Фиг. 2. Реализация на АТД `Queue`

операциите от фиг. 2, а за доказване, че операциите му запазват инвариантата на данните `front` $\leq$ `rear`, трябва да се докаже верността на предикатите, получени от спецификациите от фиг. 1 след замяна на пост-условията им с инвариантата на данните и се използват реализациите на операциите от фиг. 2.

Съществуват различни начини за верификация на програми. Тук използваме метода на преобразуващите предикати [5]. Той се основава на факта, че тройката на Хоар {`Q`}`S`{`R`} е в сила, ако е в сила импликацията `Q => Wp(S, R)`, където `Wp(S, R)` е преобразуващият предикат за програмния фрагмент `S` и пост-условието `R`.

Следните дефиниции и теорема описват някои правила за намиране на преобразуващите предикати, необходими за верифицирането на АТД Queue.

Дефиниция 1. `Wp(x = e, R) = domain(e)` $\wedge$ `R(x` $\leftarrow$ `e)`, където `domain(e)` е предикат, описващ множеството от всички състояния, в които е дефиниран изразът `e`, а `R(x` $\leftarrow$ `e)` е предикат, който се получава като в `R` променливата `x` се замени с `e`.

Дефиниция 2. `Wp(S1;S2;...;Sn, R) = Wp(S1,Wp(S2;...;Sn, R))`, където `S1, S2, ..., Sn` са произволни оператори.

Дефиниция 3. `Wp({S1;S2;...;Sn}, R) = Wp(S1;S2;...;Sn, R)`, където `S1, S2, ..., Sn` са произволни оператори.

Дефиниция 4. `Wp(if (B) S1; else S2, R) = domain(B)` $\wedge$ `(B => Wp(S1, R))` $\wedge$ `(`$\neg$`B => Wp(S2, R))`.

Теорема. *Нека за оператора* `if (B) S1; else S2` *и предикатите* `Q` *и* `R` *са в сила:* `Q => domain(B), Q ∧ B => Wp(S1, R)` *и* `Q ∧¬B => Wp(S2, R)`. *Тогава и само тогава е в сила* `Q => Wp(if (B) S1; else S2, R)`.

Верификация на реализацията на АТД Queue

Да докажем само спецификациите (1) и (3). Другите се доказват по аналогичен начин. За доказателството на спецификация 4 освен всички дефиниции се използва и теоремата.

спецификация 1

```
Wp(Initialization(), a[front..rear) = []) =
Wp(front = 0; rear=0, a[front..rear)=[])=(a[0..0)=[]) (Деф. 3, 2 и 1)
Тъй като true => (a[0..0) = []) е в сила, спецификация 1 е в сила.
```

спецификация 3

```
Wp(A = a; Front = front; Rear = rear; Push(x), (a[front..rear) =
A[Front..Rear)
   append [x])^(front = Front)^(rear = Rear+1)) =
Wp(A = a; Front = front; Rear = rear; a[rear] = x; rear = rear+1,
   (a[front..rear) = A[Front..Rear) append [x] )^(front = Front)^
        (rear = Rear+1)) =                                (Деф. 3, 2 и 1)
(a[front], a[front+1], \ldots , a[rear-1], x=a[front..rear) append [x])^
(front = front) $ \wedge $ (rear+1 = rear+1)
```

Спецификация 3 е в сила, ако е в сила импликацията
`front ≤ rear => (a[front], a[front+1], ..., a[rear-1], x = a[front..rear) append [x]) ∧ (front = front) ∧ (rear+1 = rear+1)`

Ако лявата страна на импликацията е в сила, редиците от дясната ѝ страна са коректни означения и дясната страна има стойност true. Следователнор (3) е в сила.

От доказателствата на спецификациите (1) и (3) се вижда, че инвариантата на данните на типа е в сила след изпълнението на всяка от тези операции.

Верификацията на реализация на някой АТД може да се окаже трудоемка и скъпоструваща дейност. В този случай дизайнерът на АТД може да предпочете валидация на реализацията. Да се извърши валидация на реализация на АТД означава да се докаже, че реализациите на операциите удовлетворяват спецификации, които описват свойства, различни от функционалните спецификации на операциите. Последните се наричат *валидационни свойства*. Декларират се в частта на валидационните свойства на АТД, която заменя частта на спецификациите на декларацията на АТД. В случай, че някое от валидационните свойства пропадне, следва че реализацията на АТД не е коректна.

Примери за валидационни свойства за АТД Queue са:

`{true} Initialization(); b = Empty() {b = true}`

`{front ≤ rear} p = Length(); Push(x); q = Length() {q = p+1}`

Доказателствата им могат да се осъществят по описаната по-горе методология.

Дизайнерът на АТД може да поиска да се извършат както верификация, така и валидация на реализацията на АТД. В този случай валидацията служи за повторна

проверка, а декларацията на АТД съдържа както част *спецификации*, така и част *валидационни свойства*.

Изборът на спецификациите и валидационните свойства е от изключителна важност. Желателно е те да удовлетворяват свойството *достатъчна пълнота* [4]. Последното свидетелства за това, че никое важно свойство не е останало извън спецификациите или валидационните свойства.

ЛИТЕРАТУРА

[1] П. Азълов. Как да създаваме абстрактни типове данни? *Математика и информатика*, София, 1999, **6**, 7–13.

[2] П. Азълов. Колко абстрактни са абстрактните типове данни? *Математика и информатика*, София, 1999, **5**, 10–20.

[3] П. Азълов. Плавен преход от прости типове данни към класове и обекти, *Математика и информатика*, София, 2007, No 1, 11–21.

[4] М. Бертран. Абстрактные типы данных, Москва, Интернет Университет Информационных Технологий, 2005.

[5] М. Тодорова. Програмиране на C++ – първа част, глава 12, 331–357, София, Сиела, 2002.

[6] В. Аджиев, Мифы о безопасном ПО: уроки знаменитых катастроф, Открытые системы No 6, 1998, `http://www.softwarer.ru/safety.html`.

[7] `http://en.wikipedia.org/wiki/Ariane_5_Flight_501#_note-0`.

Магдалина Василева Тодорова
ФМИ при СУ "Св. Климент Охридски"
бул. "Джеймс Баучер" 5
1126 София
e-mail: `magda@fmi.uni-sofia.bg`

VERIFICATION AND VALIDATION OF ABSTRACT DATA TYPES IMPLEMENTATIONS

Magdalina Vasileva Todorova

The paper is devoted to the abstract data types teaching in the university programming education. Special attention is paid to the verification and the validation of the abstract data types implementations.