Годишник на секция "Информатика"
Съюз на учените в България
Том 1, 2008, 40-48

Annual of "Informatics" Section
Union of Scientists in Bulgaria
Volume 1, 2008, 40-48

ДИЗАЙН НА ЕЗИК ЗА ВЕРИФИКАЦИЯ НА ПРОГРАМИ НА ОБЕКТНО-ОРИЕНТИРАНИ ЕЗИЦИ*

Магдалина Тодорова

В статията неформално и чрез примери е описан език за формална верификация на програми на обектно-ориентирани езици. Езикът има аксиоматична семантика, обогатена с техниките: програмиране чрез договор и инварианта на клас. Верификацията се свежда до доказване на теореми.

1. Мотивация

Езиците за програмиране са мощно средство за дизайн и реализация на сложен софтуер. Модерните софтуерни системи имат милиони линии код, представят стотици семантични състояния и техни преходи. Тези системи се описват трудно и най-често получените реализации са пълни с грешки. Практиката в областта на софтуерния дизайн показва необходимост от по-добри методологии и средства за дизайн, осигуряващи програмните системи да имат правилно поведение във всяко тяхно състояние. Сред най-важните изисквания за компютърните програми е тяхната правилност (тотална коректност).

Основни методи за търсене на грешки в програмите са имитационното моделиране, тестването, дедуктивният анализ и верификацията. Най-често се използва тестването, при което за допустими входни данни се проверява дали се получава правилен резултат. Чрез тестване може да се докаже присъствието на грешки, но не и тяхното отсъствие [2]. Причината е, че тестване с пълно изчерпване е практически неизползваемо. Например проста програма, която въвежда две 32-битови цели числа, намира стойността на някаква функция и извежда 32-битово цяло число, има 2^{64} (приблизително 10^{20}) възможни входа. Ако за 1 секунда може да се тестват 100 млн. случая (а това е невъзможно), пълният тест на тази проста програма ще отнеме приблизително 10^5 години.

В редица технологии като управление на транспорта, електронен бизнес, медицински системи, телефонни мрежи и др., цената на грешка в програма може да се окаже много висока. 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.$ [9, 10]. Също поради елементарна грешка медицинският ускорител Therac 25 причини 6 смъртни случая [9]. За реализирането на подобни технологии е добре да се използват формални подходи за доказване, че софтуерната система е коректна. Такива подходи са методите за верификация на програми.

През последните няколко години сред най-актуалните изследвания в областта на формалните методи на софтуерния дизайн са верификационните техники (KeY [1], Eiffel [8], ESC/Java [5], JML [6], design by contact [7] и др.).

Концепцията, наречена аксиоматична семантика, предложена от Хоар [4] преди повече от три десетилетия може да се използва за доказване на коректността на малки по обем програми. Не е приложима за верификация на сложни софтуерни системи, но може да се използва за верификация на отделни модули на тези системи. Целта на статията е да опише неформално език за софтуерен дизайн с аксиоматична семантика, позволяващ формална верификация на програми на обектно-ориентирани езици. Примерите са на езика C++, но това не е ограничение.

2. Неформално описание на езика

Направеното описание е непълно. Включва само някои основни конструкции, от които може да се получи представа за езика, използван за верификация както на обикновени процедурни програми, така и на класове на обектно-ориентирани езици.

Синтаксис

Изрази на езика	*Семантика на изразите*
Precondition P	P е предусловие
Postcondition P	P е постусловие
Class_invariant P	P е инвариантата на клас
Loop_invariant P	P е инварианта на цикъл
Limited_function f	f е ограничаваща функция на цикъл
#return == expr	expr е резултатът, върнат от обръщение към функция
#last(v)	стойността на v преди обръщението към функцията
(#sum int x; p(x); q(x))	$\sum_{x \in p(x)} q(x)$
(#prod int x; p(x); q(x))	$\prod_{x \in p(x)} q(x)$
(#forall x; p(x); q(x))	$\forall x \in p(x) : q(x)$
(#exist x; p(x); q(x))	$\exists x \in p(x) : q(x)$
P ==> Q	P => Q
P <=> Q	P $\Leftrightarrow$ Q
P && Q	конюнкция на P и Q
P || Q	дизюнкция на P и Q
! P	отрицание на P
##…##	текстът между ## и ## задава предусловия, постусловия, инварианти и ограничаващи функции

Фигура 1 дава пример за нотация на функцията fact, намираща факториел на естествено число, с помощта на изразите на езика за софтуерен дизайн.

C++ дефиниция на функция	Съответен запис на езика за софтуерен дизайн

int fact(int n)
{ int f = 1;
i = 1;
while(i < n)
{ i = i+1;
f = f*i;
}
return f;
}
	##Precondition n >= 1;
	Postcondition #return ==
	(#prod int k; k>=1&&
	k<=n; k)
	##
	int fact(int n)
	{ int f = 1;
	i = 1;
	##Loop_invariant
	i<=n && f==(#prod int k;
	1<=k&&k<=i; k);
	Limited_function
	n-i;
	##
	while(i < n)
	{ i = i+1;
	f = f*i;
	}
	#return==f;
	}

Фиг. 1. Функцията fact, зададена чрез формални спецификации

Семантика

Семантиката на езика за софтуерен дизайн е аксиоматичната семантика, към която са добавени техниките програмиране чрез договор [7] и инварианта на клас [7]. За да не усложняваме означенията, вместо за член-функция на клас, ще илюстрираме семантиката на езика чрез обикновена функция на обектно-ориентиран език.

Аксиоматичната семантика се основава на означението за условие, което е предикат, описващ състоянието на програмата (програмен фрагмент) в някакъв момент от нейното изпълнение. С програмата (програмния фрагмент) S се свързва предикатът, нарича се още тройка на Хоар, {P} S {Q}, където P и Q са предикати, изразяващи съответно предусловието и постусловието, свързани с S. Семантиката на тройката на Хоар изразява тоталната коректност на програмата (програмния фрагмент) S. В някои случаи е възможно формалното доказателство на завършването на изпълнението да е трудно или невъзможно за извеждане. Тогава се избира доказване само на частичната коректност на програмата.

Основните оператори за управление на изчислителния процес в процедурните обектно-ориентирани езици са: празен, за присвояване, блок, условен и за цикъл. За всеки от тези оператори е определено правило за извод. Използван е методът на преобразуващите предикати [3], основаващ се на факта, че тройката на Хоар {P} S {Q} е в сила, ако е в сила импликацията P => Wp(S, Q), където Wp(S, Q) е преобразуващият предикат за програмата (програмния фрагмент) S и постусловието Q. Следните дефиниции и теореми [3] определят правилата за намиране на преобразуващите предикати за операторите: празен, за присвояване, редица от

оператори, условен и за цикъл.

Дефиниция 1. Wp(празен_оператор, Q) = Q.

Дефиниция 2. Wp(x = e, Q) = domain(e) $\wedge$ Q(x $\leftarrow$ e), където domain(e) е предикат, описващ множеството от всички състояния, в които е дефиниран изразът e, а Q(x $\leftarrow$ e) е предикат, който се получава като в Q променливата x се замени с e.

Дефиниция 3. Wp(S1; S2; ...; Sn, Q) = Wp(S1, Wp(S2; ...; Sn, Q)), където S1, S2, ..., Sn са произволни оператори.

Дефиниция 4. Wp({S1; S2; ...; Sn}, Q) = Wp(S1; S2; ...; Sn, Q), където S1, S2,..., Sn са произволни оператори.

Дефиниция 5. Wp(if (B) S1; else S2, Q) = domain(B) $\wedge$ (B => Wp(S1, Q)) $\wedge$ ($\neg$B => Wp(S2, Q)).

Теорема 1. Нека за оператора if (B) S1; else S2 и предикатите P и Q са в сила: P => domain(B), P $\wedge$ B => Wp(S1, Q) и P $\wedge$ $\neg$B => Wp(S2, Q). Тогава и само тогава е в сила P => Wp(if (B) S1; else S2, Q).

В тази статия цикличните изчислителни процеси са реализирани чрез оператора while. Тъй като практически не е лесно да се приложи дефиницията на преобразуващия предикат за оператора while, ще използваме теорема, чрез която може да се провери верността на импликацията P = > Wp(while(B) S, Q).

С оператора за цикъл while(B)S; се свързват:

а) *инварианта на цикъла I* (предикат, който е в сила преди и след изпълнението на всяка стъпка на оператора while);

б) *ограничаваща функция t* (целочислена функция, явяваща се горна граница на броя на стъпките на цикъла, които остават да бъдат изпълнени). Функцията t трябва да е ограничена отдолу от 0 и при всяка стъпка от изпълнението на оператора за цикъл да намалява поне с 1.

Теорема 2. Нека предикатът I и целочислената функция t удовлетворяват условията I $\wedge$ B =>Wp(S, I), I $\wedge$ B $\Rightarrow$ t > 0 и I $\wedge$ B => Wp(t1 = t; S, t < t1), където t1 е нов идентификатор. Тогава е в сила I => Wp(while(B) S, I $\wedge$ $\neg$B).

На базата на тези дефиниции и теореми, правилата за извод P => Wp(S, Q), определящи аксиоматичната семантика на езика за софтуерен дизайн, имат вида:

P => Q	S е празният оператор
P => domain(e) && Q(x $\leftarrow$ e)	S е операторът за присвояване x = e
P => Wp({S1;S2;...;Sn}, Q) = Wp(S1;S2;...;Sn, Q) = Wp(S1,Wp(S2;...;Sn, Q))	S е операторът {S1, S2, ..., Sn}
P => domain(B) P $\wedge$ B => Wp(S1, Q) P $\wedge$ $\neg$B => Wp(S2, Q)	S е условният оператор if (B) S1; else S2
P => I I $\wedge$ $\neg$ B => Q I $\wedge$ B => Wp(St, I) I $\wedge$ B $\Rightarrow$ t > 0 I $\wedge$ B => Wp(t1 = t; St, t < t1)	S е операторът за цикъл while(B)St, където I е инвариантата, а t – ограничаващата функция на цикъла

На фиг. 2 е определена аксиоматичната семантика на функцията за намиране на факториел на естествено число, дефинирана чрез езика за софтуерен дизайн (фиг. 1).

Precondition => Wp(f = 1; i = 1; Loop_invariant)	(1)
Loop_invariant && ¬(i < n) => Postcondition	(2)
Loop_invariant &&(i < n) => Wp(i = i+1; f = f*i, Loop_invariant)	(3)
Loop_invariant &&(i < n) => Limited_function > 0	(4)
Loop_invariant &&(i < n) => Wp(t1 = Limited_function; i=i+1; f=f*i; Limited_function < t1)	(5)

Фиг. 2. Аксиоматична семантика на функцията fact

След заместване на Precondition, Postcondition, Loop_invariant и Limited_function с конкретните им предикати и намиране на преобразуващите предикати се получават импликациите:

n>=1 => 1<=n && 1 ==(#prod int k; 1<=k && k <= 1; k);

i<=n && f == (#prod int k; 1<=k && k<=i; k) && i>=n =>
f == (#prod int k; k>=1 && k<=n; k);

i<=n && f == (#prod int k; 1<=k && k<=i; k) && i<n =>
i+1<=n && f*(i+1) == (#prod int k; 1<=k && k<=i+1; k);

i<=n && f == (#prod int k; 1<=k && k<=i; k) && i<n =>
n-i>0;

i<=n && f == (#prod int k; 1<=k && k<=i; k) && i<n =>
n-(i+1) < n-i

Доказателството им доказва тоталната коректност на дефиницията на функцията fact. Конюнкцията на тези имликации може да се разглежда като теорема, доказателството на която ще верифицира формално реализацията на функцията fact. За целта може да се използва някоя от системите за доказателство на теореми като KeY, COQ, HOL, ACL2 и др.
Следва фрагмент от доказателството на теоремата за функцията fact чрез HOL[11].

```
- open arithmeticLib;
- val ARW_TAC = RW_TAC arith_ss;
- val INV_FACT = store_thm
("INV_FACT", ``!a b. (a = b) ==> (FACT a = FACT b)``,
 ARW_TAC[]);
- g `( n >= 1 ==> n >= 1 /\ (FACT (SUC 0) = 1)) /\
(i <= n /\ (f = FACT i) /\ n <= i ==> (f = FACT n)) /\
```

```
        (i <= n /\ (f = FACT i) /\ i < n ==>
        SUC i <= n /\ (FACT (SUC i) = SUC i * f)) /\
        (i <= n /\ (f = FACT i) /\ i < n ==> n - i > 0) /\
        (i <= n /\ (f = FACT i) /\ i < n ==>
    n - SUC i < n - i)`;
        - e (ARW_TAC[FACT, INV_FACT]);
        OK..
        > val it =
        Initial goal proved.
        ...
```

Използва се FACT, дефинирана в библиотеката arithmeticLib по следния начин:

```
        (FACT 0 = 1)
        (!n. FACT (SUC n) = SUC n * FACT n)
```

Ако функцията е член-функция на клас доказателството на нейната тотална коректност е подобно на показаното по-горе. Ако се наложи доказване само на частичната коректност, в дефиницията на функцията чрез езика за софтуерен дизайн трябва да се пропусне задаването на Limited_function. Аксиоматичната семантика в този случай не съдържа импликациите (4) и (5) от фиг. 2.

Следователно, описаният език за софтуерен дизайн съдържа средства за дефиниране на предусловия, постусловия, инварианти на клас и на цикъл, ограничаващи функции и проверява дали програма е тотално (или частично) коректна.

3. Коректност на обектно-ориентирани програми

Обектно-ориентираните програми са съвкупности от класове. Когато реализира такъв вид програми, програмистът трябва да има средства, чрез които да може формално да провери дали цялата програма извършва това, за което е предназначена. Към описания по-горе език за софтуерен дизайн се добавят още техниките: програмиране чрез договор и инварианта на клас, необходими за верификацията на класове и йерархии от класове.

Програмиране чрез договор

Обектите и методите на клас най-често се използват от методите на други класове. Последните се наричат класове клиенти. Идеята е, между класа и неговите клиенти да има правила (договори), съгласно които клиентът трябва да осигури подходящи стойности на параметрите, а класът доставчик да намери и върне търсения резултат [7]. Клиентът трябва да гарантира, че ще са в сила някои предусловия преди да извика методите на класа. Класът доставчик гарантира от своя страна, че ще са в сила определени постусловия след изпълнението на съответните методи.

Програмирането чрез договор е средство, което съдържа “задълженията” и “правата” между всяка двойка клиент/доставчик по време на живота на софтуерната система. За една двойка клиент/доставчик “задълженията” и “правата” се задават чрез матрица от вида:

	Задължения	*Права*
Клиент	Да удовлетворява предусловието	Да получи търсения резултат
Доставчик	Да удовлетворява постусловието	Да получи вход, който удовлетворява предусловието

За методите на класовете *договорите* се задават чрез указване на предусловията и постусловията със спецификации от вида:

```
## Method_Precondition
        …
 Method_Postcondition
    …
##
```

Задаването на договорите предшества декларациите (дефинициите) на методите на класовете. Фиг. 3 описва класа Queue, реализиращ АТД опашка от цели числа, чрез езика за софтуерен дизайн. Описанието съдържа договорите за всеки метод на класа.

Инварианта на клас

Всеки обект на клас трябва да удовлетворява множество от свойства, които го свързват с класа. Тези свойства се определят чрез предикат, който се нарича *инварианта* на класа. За всеки клас се дефинира инварианта. Тя трябва да е в сила след изпълнението на всеки конструктор и преди, и след изпълнението на всеки метод на класа. Задава се чрез спецификацията

```
## Class_invariant
    ...
##
```

За класа Queue (фиг. 3), спецификацията:

```
## Class_invariant
    front ≤ rear
##
```

задава инвариантата му. Тъй като опашката е представена чрез редицата от цели числа a_{front}, $a_{front+1}$, …, a_{rear-1} (съкратено a[front..rear)), инвариантата front ≤ rear на класа изразява съществуването на редицата. Ако front == rear, опашката е празна.

Да е тотално коректен клас на обектно-ориентирана програма означава да са тотално коректни всички негови методи (да са в сила всички клиент/доставчик-договори) и освен това да е в сила инвариантата на класа преди и след изпълнението на всеки метод на класа. Ще дадем и формален запис на тази дефиниция.

```
class Queue
{ private:
   ## Class_invariant
      front ≤ rear
   ##
```

```
void Push(int x)
{ a[rear] = x;
  rear = rear + 1;
}
## Method_Precondition
```

```
  int a[size];
  int front, rear;
public:
  ## Method_Precondition
    true
    Method_Postcondition
    a[front..rear) == []
  ##
  Queue()
  { front = 0; rear = 0;
  }
  ## Method_Precondition
          front ≤ rear
    Method_Postcondition
    #return ==
        (front == rear)
  ##
  bool Empty()
  { return front == rear;
  }
  ## Method_Precondition
          front ≤ rear
    Method_Postcondition
    (a[front..rear) ==
#last(*this).a append [x])
 &&
(rear == #last(*this).rear+1)
  ##
  front < rear
  Method_Postcondition
  (#return ==
  a[#last(*this).front]) &&
  (a[front..rear) ==
#last(*this).a[#last(*this).
front+1..#last(*this).rear) &&
(front == #last(*this).front+1)
##
int Pop()
{ int x;
 if (rear – front > 0)
   { x = a[front];
     front = front + 1;
   }
 else x = -1;
   return x;
}
 ## Method_Precondition
  front ≤ rear
  Method_Postcondition
  (#return == rear – front) &&
  (*this == #last(*this))
##
int Length()
{ return rear – front;
}
};
```

Фиг. 3. Класът Queue, зададен чрез формални спецификации

Нека P_i и Q_i са предусловието и постусловието на i-я конструктор C_i или i-я метод M_i на клас, а инвариантата на класа, от който са тези методи, е Class_invariant. Класът е тотално коректен относно свързаните с него твърдения, ако:

- за всеки конструктор C_i е в сила

$\{ P_i \} C_i \{ Q_i \wedge \text{Class\_invariant} \}$

- за всеки метод M_i е в сила

$\{ P_i \wedge \text{Class\_invariant} \} M_i \{ Q_i \wedge \text{Class\_invariant} \}$.

Чрез техниката за верификация на функция, описана в предишната точка, може да се докаже верността на всички тройки на Хоар за класа Queue.

4. Заключение

Развитието на технологиите за формален софтуерен дизайн е сред най-перспективните направления на софтуерното инженерство. В редица софтуерни технологии използването им се налага. Голяма част от програмистите не са обучени да използват средствата на математическата логика при разработката на програми. Тези програмисти се нуждаят от съвременни софтуерни техники за формален дизайн, които да им помогнат при формалната верификация на софтуерните системи. И най-накрая, използването на формалните методи повишава качеството на софтуерния дизайн и съкращава времето за реализацията му.

Литература

[1] W. Ahrendt, B. Beckert, R. Hahnle, P. Ruummer, P. Schmitt, Verifying Object-Oriented Programs with KeY: A Tutorial, 5th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands, November 2006, LNCS 4709, Springer-Verlag.

[2] E.W.Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[3] D. Gries, The Science of Programming, Springer-Verlag, Berlin and New York, 1981.

[4] C.A.R. Hoare, Proof of Correctness of Data Representations, Acta Informatica, Vol. 1, N 4, 1972 (pp. 271-281).

[5] C. Flanagan, Rustan M., Extended Static Checking for Java, PLDI'02, June 17-19, 2002, Berlin, Germany.

[6] G. T. Leavens, Y. Cheon, How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (eds.), Formal Methods for Components and Objects, pages 262-284. Volume 2852 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 2003.

[7] B. Meyer. Applying "design by contract", Computer 25(10), 40-51, 1992.

[8] B. Meyer. Eiffel: The Language. Prentice Hall, New York, 1992.

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

[10] http://portal.acm.org/citation.cfm?doid=251880.251992 Dowson, Mark, "The Ariane 5 Software Failure". Software Engineering Notes, Volume 22, Issue 2 (March 1997), page 84

[11] http://isabelle.in.tum.de/documentation.html.