УДК 519.86:347.464

ИНТЕРВАЛЬНОЕ ОЦЕНИВАНИЕ ПАРАМЕТРОВ ДЛЯ ОПРЕДЕЛЕНИЯ НАДЁЖНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В.Ю. Дубницкий[1], А.М. Кобылин[2], О.А. Кобылин[3]

[1] ХИБД УБД, г. Харьков, Украина, valeriy_dubn@mail.ru;
[2] ХИБД УБД, г. Харьков, Украина, kobilin@khibs.edu.ua;
[3] ХНУРЭ, м. Харьков, Украина, kblin@ kture.kharkov.ua;

Предложен способ оценивания параметров, определяющих надёжность программного обеспечения. Для этого выполнено исследование и решение простейших алгебраических уравнений, коэффициенты которых принадлежат множеству интервальных чисел. Разработанное программное средство позволяет пользователю проводить вычисления, выбирая из соответствующих списков названия переменных, совпадающих с названиями показателей, вводить исходные данные, выбирать тип операции, выполнять расчеты и, в случае потребности, сохранять их на листе электронной таблицы и представлять результаты в графическом виде.

ИНТЕРВАЛЬНЫЕ ЧИСЛА, НАДЕЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, ОБРАТНАЯ ПОЛЬСКАЯ ЗАПИСЬ, СТЕКОВЫЙ ИНТЕРВАЛЬНЫЙ КАЛЬКУЛЯТОР, МОДЕЛЬ ХОЛСТЕДА, МОДЕЛИ МИЛСА

Введение

Одной из существенных задач в разработке программного обеспечения является его тестирование на наличие ошибок (так называемых багов). Это обусловлено тем, что на качество программного продукта может влиять достаточно много факторов. Они варьируют в зависимости от назначения программного продукта, сложности поставленной задачи и объемности. Очень часто их характеристики имеют высокий уровень нестохастической неопределенности. Высокая надежность, предъявляемая к программным системам критического применения, требует проведения большого числа испытаний для достоверного определения их безотказной работы. В [1] приведена оценка необходимого количества испытаний, которая представлена в виде:

$$n = \frac{\lg(1-\beta)}{\lg(1-p)}, \qquad (1)$$

где: n – необходимое количество испытаний; β – доверительная вероятность, p – верхнее допустимое значение вероятности отказа.

Результаты соответствующих вычислений приведены в табл. 1.

Таблица 1

Определение количества испытаний программной системы при заданной доверительной вероятности и вероятности отказа

Вероятность отказа p	Уровень доверительной вероятности β			
	0,95	0,975	0,99	0,995
$1\cdot10^{-2}$	299	367	458	528
$1\cdot10^{-3}$	2995	3687	4603	5296
$1\cdot10^{-4}$	29956	36887	46049	52981
$1\cdot10^{-5}$	999572	368886	460515	529829
$1\cdot10^{-6}$	2995731	3688878	4605158	5298315

Из этой таблицы видно, что при испытании программных систем в режиме реального времени и при высоких требованиях к их безотказности количество испытаний достигает сотен тысяч и даже миллионов, а длительность испытаний может быть соизмеримой с месяцами или даже годами.

В этом случае адекватным математическим аппаратом для количественного анализа результатов тестирования компьютерных программ служит аппарат интервальных вычислений [2, 3].

Целью работы является разработка метода, позволяющего формулировать требования к параметрам, определяющим надёжность программного обеспечения при нестохастической их неопределённости. Подобные ситуации имеют место на различных стадиях предварительного проектирования сложных программных продуктов.

1. Анализ предметной области и постановка задачи

В связи с необходимостью решения обратной задачи – подбора параметров программы, гарантирующих ее необходимую надежность, далее используем систему аксиом $АК_2$, приведенную в работе [3] .

Пусть символ «o» означает одну из операций +, -, *, / имеющих традиционный смысл. Используя алгебраическую символику запишем, что

$$o \in \{+,-,\cdot,/\}. \qquad (2)$$

Тогда для двух интервальных чисел $[A_1],[A_2]$ справедливо условие:

$$[A_1]\, o\, [A_2] = (\min U, \max U), \qquad (3)$$

где

$$U = \left((a_1,a_2)\ o\ \left(b_1,b_{b_2}\right)\right) = \\ = (a_1\ o\ b_1),(a_1\ o\ b_2),(a_2\ o\ b_1),(a_2\ o\ b_2). \qquad (4)$$

Сравним выполнение операции $(-)$ в системах аксиом AK_1, приведенной в [2], и AK_2, приведенной в [3], используя геометрическую иллюстрацию (рис. 1).

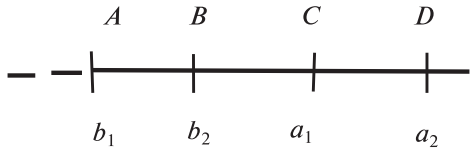

Рис. 1. Геометрическая иллюстрация выполнения операции $(-)$

Очевидно, что:

$$DA = a_2 - b_1;\ CA = a_1 - b_1;\ DB = a_2 - b_2;$$
$$CB = a_1 - b_2.$$

Тогда отрезком наибольшей длины будет отрезок DA, наименьшей – отрезок CB. Таким образом, показано совпадение этих двух систем аксиом, потому что условия (2, 3, 4)совпадают с условием (1).

Следуя работе [3], назовем интервалы $[A]=(a_1,a_2)$ и $\left[\overline{A}\right]=(a_2,a_1)$ сопряженными.

В этой же работе показано, что решения алгебраических уравнений, коэффициенты которых есть интервальные числа, могут быть получены в следующем виде:

Уравнение I типа:

$$[A]+[X]=[B], \quad (5)$$

тогда

$$[X]=[B]-[A]. \quad (6)$$

Уравнение II типа:

$$[A]\cdot[X]=[B],\ 0\notin[A], \quad (7)$$

тогда

$$[X]=\frac{[B]}{[A]}. \quad (8)$$

Уравнение III типа:

$$[A]\ [X]+[B]=[C], \quad (9)$$

тогда

$$[X]=\frac{[C]-[\overline{B}]}{[\overline{A}]} \quad (10)$$

при условии, что $0\notin[A]$.

Приведем численные примеры.

Рассмотрим решение уравнения типа I.

[X] + [5; 7] =[14; 18]

[X]=[14; 18] – [7; 5]

U = (7; 11; 9; 13)

min U =7, max U = 13

[Y] = [7,13].

Рассмотрим решение уравнения типа II.

$$[X]\ [5;\ 7] = [14;\ 18].$$

$$[X]=\frac{[14;\ 18]}{[7;\ 5]};$$

$$U=\left(2;\ \frac{14}{5};\ \frac{18}{7};\ \frac{18}{5}\right),\quad \min U=2;\quad \max U=\frac{18}{5}$$

$$[Y]=\left[2;\ \frac{18}{5}\right].$$

Рассмотрим решение уравнения типа III.

Пусть [A] = [2; 5]; [B] = [4; 9]; [C] = [15; 23].

Тогда:

$$[2;\ 5]\ *\ [X] + [4;\ 9] = [15;\ 23]$$

В соответствии с (28)

$$[X]=\frac{[15;\ 23]-[9;\ 4]}{[5;\ 2]}=\left[\frac{6}{5};\ \frac{19}{2}\right].$$

Следовательно, при решении прямых задач применение аксиом AK_1 и AK_2 приводит к одинаковым результатам, при решении обратных задач следует использовать систему аксиом AK_2.

2. Модели оценки надежности программного обеспечения

Далее рассмотрим основные модели оценки надежности программного обеспечения, приведенные в работе [4], но для их анализа используем аппарат интервальных вычислений.

Модель Холстеда

$$N_{\text{ошибок}}=\frac{V}{E_{\text{критическое}}}, \quad (11)$$

где: V – объем программы; $E_{\text{критическое}}$ – эмпирическая постоянная $E = 2*10^3 \ldots 5*10^3$; $N_{\text{ошибок}}$ – количество ошибок в программе.

В интервальном виде эта модель имеет вид:

$$\left[N_{\text{ошибок}}\right]=[A]\ [V], \quad (12)$$

где

$$\left[N_{\text{ошибок}}\right]=\left[2\cdot10^{-4};5\cdot10^{-4}\right][V], \quad (13)$$

то есть получено уравнение вида I.

На рис. 2 приведена Web-форма решения обратной задачи оценки надежности программного обеспечения с использованием интервального калькулятора модели Холстеда.

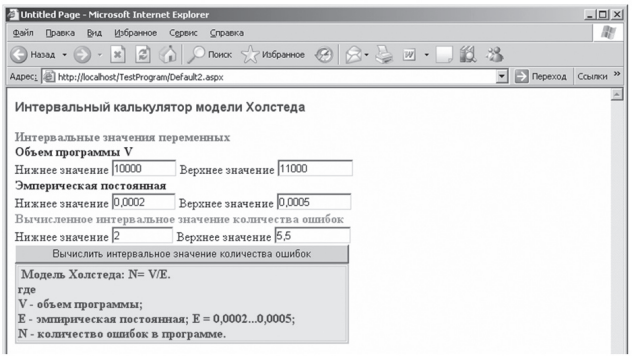

Рис. 2. Web-форма решения обратной задачи оценки надежности программного обеспечения с использованием интервального калькулятора модели Холстеда

Модель Миллса

$$N_{ош} = \frac{N_{отм} - N_{внес}}{N_{внесв}}, \qquad (14)$$

где: $N_{ош}$ – количество собственных дефектов в программе; $N_{отм}$ – количество выявленных собственных дефектов в программе; $N_{внес}$ – количество внесенных дефектов; $N_{внесв}$ – количество выявленных внесенных дефектов.

Для решения обратной задачи возможны три варианта: решение относительно $N_{отм}$, относительно $N_{внес}$, относительно $N_{внесв}$. Однако в любом случае задача сводится к уравнению вида II. Его варианты приведены в табл 2.

Таблица 2

Условия задачи по определению параметров модели Миллса

$[A]$	$[X]$	$[B]$
$[N_{отм}] / [N_{внесв}]$	$[N_{внес}]$	$[N_{ош}]$
$[N_{внес}] / [N_{внесв}]$	$[N_{отм}]$	$[N_{ош}]$
$[N_{отм}] / [N_{внес}]$	$[X] = [1; 1]\,[N_{внес}]$ $[N_{внес}] = [X]^{-1}$	$[N_{ош}]$

На рис. 3 показана Web-форма решения прямой и обратной задач оценки надежности программного обеспечения с использованием интервального калькулятора модели Миллса.

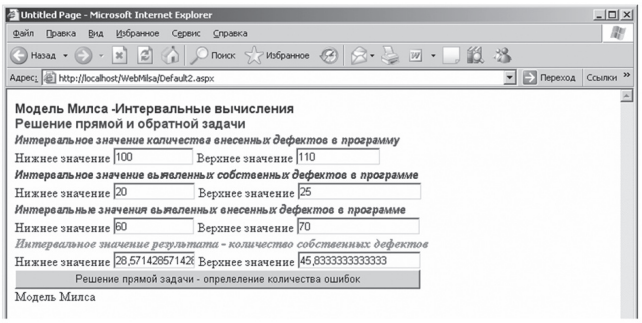

Рис. 3. Web-форма решения прямой и обратной задач оценки надежности программного обеспечения с использованием интервального калькулятора модели Миллса

Модель фирмы IBM

$$N = \alpha \cdot MUM + \gamma \cdot UM\ , \qquad (15)$$

где: α, γ – эмпирические постоянные, $\alpha = 23$; $\gamma = 2$; MUM – количество многократно исправляемых модулей (более десяти раз); UM – количество исправляемых модулей (не более десяти раз).

В свою очередь

$$UM = 0{,}9\ NM + 0{,}15\ CUM, \qquad (16)$$

$$MUM = 0{,}15\,UM + 0{,}06\,CUM, \qquad (17)$$

где NM – количество модулей, CUM – количество старых исправляемых модулей.

Подставив (16) и (17) в (15), в интервальном виде получим уравнение

$$N = [3{,}45; 3{,}45]*[MUM]+[1{,}8; 1{,}8]*[NM] + \\ +[1{,}68; 1{,}68]*[CUM]. \qquad (18)$$

Пусть

$$[3{,}45; 3{,}45] = [F];\ \ [1{,}8; 1{,}8] = [G]; \\ [1{,}68; 1{,}68] = [Q].$$

Тогда

$$[N] = [F]*[NUM] + [G]*[NM] + [Q]*[CUM]. \qquad (19)$$

При решении этого уравнения относительно одного из параметров получим уравнение типа III.

Варианты постановки задачи приведены в табл. 3.

Таблица 3

Условия задачи по определению параметров модели фирмы IBM

$[A]$	$[X]$	$[B]$	$[C]$
$[A]$	$[M]$	$[B]*[NM] + [C]*[CUM]$	$[N]$
$[B]$	$[NM]$	$[A]*[M] + [C]*[CUM]$	$[N]$
$[C]$	$[CUM]$	$[A]*[M] + [B]*[NM]$	$[N]$

3. Результаты исследования

Программирование калькулятора в режиме обратной польской записи является существенным методом повышения надежности программ является сокращение ее объем, в частности, в соответствии с метрикой Холстеда, количество ошибок в программе после окончательной ее разработки определяется по формуле (11). Доказательство данного метода приведено в работе [5]. Для сокращения объема программ предлагается использование так называемой обратной польской записи (ОПЗ). Сравнительный анализ объема программ, разработанных в обычной нотации и ОПЗ, показал, что для формул, в состав которых входит более 5 операндов, экономия составляет примерно 17%.

Для реализации ОПЗ использовался стек, являющийся наиболее важной из структур данных. Эти структуры встречаются в программировании буквально на каждом шагу, в самых разнообразных ситуациях. Особенно интересен стек, который имеет самые неожиданные применения. В свое время при разработке серии ЭВМ IBM 360 в начале 70-х годов XX века фирма IBM совершила драматическую ошибку, не предусмотрев аппаратную реализацию стека. Эта серия содержала много других неудачных решений, но, к сожалению, была скопирована в Советском Союзе под названием ЕС ЭВМ (Единая Серия), а все собственные разработки были приостановлены. Это отбросило советскую промышленность на много лет назад в области разработки компьютеров.

Стек — самая популярная и, пожалуй, самая важная структура данных в программировании. Стек представляет собой запоминающее устройство, из которого элементы извлекаются в порядке,

обратном их добавлению. Это как бы неправильная очередь, в которой первым обслуживают того, кто встал в нее последним. В программистской литературе общепринятыми являются аббревиатуры, обозначающие дисциплину работы стека. Дисциплина работы стека обозначается LIFO, последним пришел — первым уйдешь (Last In First Out).

Стек можно представить в виде трубки с подпружиненым дном, расположенной вертикально. Верхний конец трубки открыт, в него можно добавлять, или, как говорят, заталкивать элементы. Общепринятые английские термины в этом плане очень красочны, операция добавления элемента в стек обозначается push, в переводе "затолкнуть, запихнуть". Новый добавляемый элемент проталкивает элементы, помещенные в стек ранее, на одну позицию вниз. При извлечении элементов из стека они как бы выталкиваются вверх, по-английски pop ("выстреливают").

Примером стека может служить стопка бумаг на столе, стопка тарелок и тому подобное. Отсюда произошло название стека, что по-английски означает стопка. Тарелки снимаются со стопки в порядке, обратном их добавлению. Доступна только верхняя тарелка, то есть тарелка на вершине стека. Хорошим примером будет также служить железнодорожный тупик, в который можно составлять вагоны.

Стек применяется довольно часто, причем в самых разных ситуациях. Объединяет их следующая цель: нужно сохранить некоторую работу, которая еще не выполнена до конца, при необходимости переключения на другую задачу. Стек используется для временного сохранения состояния не выполненного до конца задания. После сохранения состояния компьютер переключается на другую задачу. По окончании ее выполнения состояние отложенного задания восстанавливается из стека, и компьютер продолжает прерванную работу.

Почему именно стек используется для сохранения состояния прерванного задания? Предположим, что компьютер выполняет задачу *A*. В процессе ее выполнения возникает необходимость выполнить задачу *B*. Состояние задачи *A* запоминается, и компьютер переходит к выполнению задачи *B*. Но ведь и при выполнении задачи *B* компьютер может переключиться на другую задачу *C*, и нужно будет сохранить состояние задачи *B*, прежде чем перейти к *C*. Позже, по окончании *C* будет вначале восстановлено состояние задачи *B*, затем, по окончании *B* — состояние задачи *A*. Таким образом, восстановление происходит в порядке, обратном сохранению, что соответствует дисциплине работы стека.

Стек позволяет организовать рекурсию, то есть обращение подпрограммы к самой себе либо непосредственно, либо через цепочку других вызовов. Пусть, например, подпрограмма *A* выполняет алгоритм, зависящий от входного параметра *X* и, возможно, от состояния глобальных данных. Для самых простых значений *X* алгоритм реализуется непосредственно. В случае более сложных значений *X* алгоритм реализуется как сведение к применению того же алгоритма для более простых значений *X*. При этом подпрограмма *A* обращается сама к себе, передавая в качестве параметра более простое значение *X*. При таком обращении предыдущее значение параметра *X*, а также все локальные переменные подпрограммы *A* сохраняются в стеке. Далее создается новый набор локальных переменных и переменная, содержащая новое (более простое) значение параметра *X*. Вызванная подпрограмма *A* работает с новым набором переменных, не разрушая предыдущего набора. По окончании вызова старый набор локальных переменных и старое состояние входного параметра *X* восстанавливаются из стека, и подпрограмма продолжает работу с того места, где она была прервана.

На самом деле даже не приходится специальным образом сохранять значения локальных переменных подпрограммы в стеке. Дело в том, что локальные переменные подпрограммы (то есть ее внутренние, рабочие переменные, которые создаются в начале ее выполнения и уничтожаются в конце) размещаются в стеке, реализованном аппаратно на базе обычной оперативной памяти. В самом начале работы подпрограмма захватывает место в стеке под свои локальные переменные, этот участок памяти в аппаратном стеке называют обычно блок локальных переменных или по-английски frame ("кадр"). В момент окончания работы подпрограмма освобождает память, удаляя из стека блок своих локальных переменных.

Кроме локальных переменных, в аппаратном стеке сохраняются адреса возврата при вызовах подпрограмм. Пусть в некоторой точке программы *A* вызывается подпрограмма *B*. Перед вызовом подпрограммы *B* адрес инструкции, следующей за инструкцией вызова *B*, сохраняется в стеке. Это так называемый адрес возврата в программу *A*. По окончании работы подпрограмма *B* извлекает из стека адрес возврата в программу *A* и возвращает управление по этому адресу. Таким образом, компьютер продолжает выполнение программы *A*, начиная с инструкции, следующей за инструкцией вызова. В большинстве процессоров имеются специальные команды, поддерживающие вызов подпрограммы с предварительным помещением адреса возврата в стек и возврат из подпрограммы по адресу, извлекаемому из стека. Обычно команда вызова называется call, команда возврата — return.

В стек помещаются также параметры подпрограммы или функции перед ее вызовом. Порядок их помещения в стек зависит от соглашений, при-

нятых в языках высокого уровня. Так, в языке Си или C++ на вершине стека лежит первый аргумент функции, под ним второй и так далее. В Паскале всё наоборот, на вершине стека лежит последний аргумент функции. (Поэтому, кстати, в Си возможны функции с переменным числом аргументов, такие, как printf, а в Паскале нет.). В языке C# для работы со стеком создан специальный класс Stack, свойства и методы которого представлены в табл. 4 и 5.

Таблица 4

Свойство класса Stack

Имя	Описание
Count	Возвращает число элементов в стеке

Таблица 5

Методы Stack

Имя	Описание
Pop	Возвращает элемент с вершины стека, одновременно удаляя его
Push	Добавляет элемент на вершину стека
Peek	Возвращает верхний элемент стека, не удаляя его

В работе рассматривается создание специализированного интервального калькулятора, реализующего обратную польскую запись с использованием стека для выполнения финансовых расчетов.

Стековый интервальный калькулятор и обратная польская запись формулы.

В 1920 г. польский математик Ян Лукашевич предложил способ записи арифметических формул, не использующий скобок. В привычной нам записи знак операции записывается между аргументами, например сумма чисел 2 и 3 записывается как 2 + 3. Ян Лукашевич предложил две другие формы записи: префиксная форма, в которой знак операции записывается перед аргументами, и постфиксная форма, в которой знак операции записывается после аргументов. В префиксной форме сумма чисел 2 и 3 записывается как + 2 3, в постфиксной — как 2 3 +. В честь Яна Лукашевича эти формы записи называют прямой и обратной польской записью.

В польской записи скобки не нужны. Например выражение

(2+3)*(15-7)

записывается в прямой польской записи как

* + 2 3 - 15 7,

в обратной польской записи — как

2 3 + 15 7 - *.

В стековом интервальном калькуляторе с ОПЗ для вычисления выражения

[2;3]+[3;4]-[1;3]

необходимо записать

2 3 3 4 + 1 3 -

Если прямая польская запись не получила большого распространения, то обратная оказалась чрезвычайно полезной. Неформально преимущество обратной записи перед прямой польской записью или обычной записью можно объяснить тем, что гораздо удобнее выполнять некоторое действие, когда объекты, над которыми оно должно быть совершено, уже даны.

Обратная польская запись формулы позволяет вычислять выражение любой сложности, используя стек как запоминающее устройство для хранения промежуточных результатов. Обычные модели калькуляторов не позволяют вычислять сложные формулы без использования бумаги и ручки для записи промежуточных результатов. В некоторых моделях есть скобки с одним или двумя уровнями вложенности, но более сложные выражения вычислять невозможно. Также в обычных калькуляторах трудно понять, как результат и аргументы перемещаются в процессе ввода и вычисления между регистрами калькулятора. Калькулятор обычно имеет регистры *X*, *Y* и регистр памяти, промежуточные результаты каким-то образом перемещаются по регистрам, каким именно — запомнить невозможно.

В отличие от других калькуляторов, устройство стекового калькулятора вполне понятно и легко запоминается. Калькулятор имеет память в виде стека. При вводе числа оно просто добавляется в стек. При нажатии на клавишу операции, например на клавишу +, аргументы операции сначала извлекаются из стека, затем с ними выполняется операция, наконец, результат операции помещается обратно в стек. Таким образом, при выполнении операции с двумя аргументами, например сложения, в стеке должно быть не менее двух чисел. Аргументы удаляются из стека и на их место записывется результат, то есть при выполнении сложения глубина стека уменьшается на единицу. Вершина стека всегда содержит результат последней операции и высвечивается на дисплее калькулятора.

Реализация стекового интервального калькулятора на C#.

Рассмотрим проект, реализующий стековый интервальный калькулятор на C#. Такая программа весьма полезна, поскольку позволяет проводить вычисления, не прибегая к записи промежуточных результатов на бумаге.

Программа состоит из трех файлов: “Form1.cs”, “Form2.cs” и “Program.cs”. Первые два файла реализуют стек вещественных чисел, эта реализация уже рассматривалась ранее. Файл “Form1.cs” реализует интервальный стековый калькулятор на базе ОПЗ. Файл Form2.cs реализует финансовые вычисления структур данных типа стек на ОПЗ. Файл Program.cs – основной файл проекта.

Ниже приведено содержимое двух фрагментов файла “Form1.cs”. Функция main, описанная в этом файле, организует диалог с пользователем в режиме команда-ответ. Пользователь может ввести число с клавиатуры, это число просто добавляется в стек. При вводе одного из четырех знаков арифметических операций +, -, *, / программа извлекает из стека два числа, выполняет указанное арифметическое действие над ними и помещает результат обратно в стек. Значение результата отображается также на дисплее. Кроме арифметических операций, пользователь может ввести название одной из стандартных функций: exp(x), ln, log (десятичный логарифм), sqrt(x). При этом программа извлекает из стека аргумент функции, вычисляет значение функции и помещает его обратно в стек. При желании список стандартных функций и возможных операций можно расширить. Каждая команда стекового калькулятора реализуется с помощью отдельной функции. Например сложение реализуется с помощью функции AddIntOPN():

```
   private void AddIntOPN_Click(object sender,
EventArgs e)
   {
     if (EntryInProgress)
         InitializeRegisterFromDisplay();
       if (RegStack1.Count >= 2)
    {
     double op4 = (double)RegStack1.Pop();
     double op3 = (double)RegStack1.Pop();
     double op2 = (double)RegStack1.Pop();
     double op1 = (double)RegStack1.Pop();
     double a = op1 + op3;
     double b = op2 + op4;
     string display1 = a.ToString(FormatString);
     string display2 = b.ToString(FormatString);
     textBox3.Text = display1;
     textBox4.Text = display2;
     RegStack1.Push(a);
     RegStack1.Push(b);
     Display1.Text = " ";
     Display1.Text = " ";
    }
   }
```

В начале функции проверяется, что глубина стека не меньше двух. В противном случае, выдается сообщение об ошибке, и функция завершается. Далее из стека извлекаются операнды y и x операции вычитания. Элементы извлекаются из стека в порядке, обратном их помещению в стек, поэтому y извлекается раньше, чем x. Затем вычисляется разность $y - x$, ее значение помещается обратно в стек и печатается на дисплее, для печати вершины стека вызывается функция display.

Приведем начальный фрагмент программы, описывающий объявление экземпляров класса Stack и переменных.

```
namespace Калькулятор_Интервальный_ОПЗ
{
  public partial class IntSQRT : Form
  {
   public IntSQRT()
    {
     InitializeComponent();
      }
       Stack RegStack1 = new Stack();
       Stack RegStack2 = new Stack();
       string FormatString = "f2";
       const int MaxChars = 21;
       private bool FixPending = false;
       private bool DecimalInString = false;
       private bool EntryInProgress = false;
       private void
       InitializeRegisterFromDisplay()
        {
   double x = (Display1.Text.Length == 0 || Display1.
Text == ",") ? 0.0 : Convert.ToDouble(Display1.
Text);
   RegStack1.Push(x);
   double x1 = (Display2.Text.Length == 0 || Display2.
Text == ",") ? 0.0 : Convert.ToDouble(Display2.
Text);
   RegStack1.Push(x1);
    }
```

В табл. 6 и 7 приведены сравнительные результаты для выражений различной степени сложности на обычном калькуляторе, калькуляторе с ОПЗ и интервальном калькуляторе с ОПЗ.

При выполнении расчетов для начисления простых и сложных процентов, количество операций сократилось на две. При выполнении расчетов на интервальном стековом калькуляторе количество операций сократилось на три. Таким образом с повышением сложности расчетов эффект от использования ОПЗ увеличивается.

Таблица 6

Формулы начисления простых и сложных процентов

Обычный калькулятор			
Действия на обычном калькуляторе		Действия на калькуляторе с ОПЗ	
$S=P(1+rt)$	$S=P(1+r)^t$	$S=P(1+rt)$	$S=P(1+r)^t$
0,1	1	0,1	0,1
*	+	Ввод	Ввод
5	0,1	5	1
=	=	*	+
+	x^y	1	5
1	5	+	a^x
=	=	1000	1000
*	*	*	* (1610,51)
1000	1000		
=	= (1610,51)		

Таблица 7

Формулы начисления простых и сложных процентов

Интервальный калькулятор	
Действия на интервальном калькуляторе с ОПЗ	
$[S_н; S_к] = [P_н; P_к]\cdot(1+[r_н; r_к]\cdot[t_н; t_к])$	$[S_н; S_к] = [P_н; P_к]\cdot(1+[r_н; r_к])^{[t_н; t_к]}$
[0,1; 0,1] Ввод	[1;1] Ввод
[5,0; 5,1] Ввод	[0,1;0,1] Ввод
Умножение интервальное	Сложение интервальное
[1;1] Ввод	[5;5,1] Ввод
Сложение интервальное	Возведение в степень интервальное *a^x*
[1000;1100] Ввод	[1000;1100] Ввод
Умножение интервальное [1500;1717,1]	Умножение интервальное [1610,51;1873,01]

S – накопленная сумма, P – сумма вклада, r – процентная ставка, t – срок вклада.

$P = 1000$, $r = 0{,}1$, $t = 5$.

S – накопленная сумма ренти, r – размер члена ренти, i – годовая процентная ставка, n – *срок* ренти (в годах)

Выводы

Показана эффективность применения интервальных вычислений для решения задач, связанных с тестирование программ в условиях, в которых применение традиционных методов прогнозирования невозможно или усложнено отсутствием сведений о статистических свойствах переменных.

Приведены сведения о специализированном программном стековом калькуляторе, который реализует правила интервальной арифметики для оценивания эффективности тестирования компьютерных программ.

Показано, что применение обратной польской записи позволяет сократить объем программы, следовательно, повысить надежность.

В связи с увеличением объема финансовых вычислений и в целях повышения ответственности за их результаты к перспективным направлениям дальнейших работ следует отнести организацию схемы дистанционного доступа к вычислительным ресурсами и создание специализированных сетей, предназначенных для анализа финансовых данных.

Список литературы: 1. *Вентцель, Е.С.* Теория вероятностей [Текст] : учебник / Е.С. Вентцель. – М .: Гос. изд-во физмат. лит-ры, 1962. –560 с. **2.** *Алефельд, Г.* Введение в интервальные вычисления [Текст] / Г. Алефельд, Ю. Херцбергер; – М.: Мир, 1987. – 259 с. **3.** *Алтунин, А.Е.* Модели и алгоритмы принятия решений в нечетких условиях [Текст] / А.Е. Алтунин, М. В. Семухин; – Тюмень: Изд. ТГУ, 2000. – 352 с. **4.** *Харченко, В.С.* Методы моделирования и оценки качества и надежности программного обеспечения [Текст] / В.С. Харченко, В.В. Скляр, О.М. Тарасюк; – Учеб. пособие. – Харьков: Нац. аэрокосм. ун-т "Харьк. авиац. ин-т", 2004. – 159 с. **5.** *Пайчун, Б.П.* Оценка надежности программного обеспечения [Текст] / Б.П.Пайчун, Р.М. Юсупов; – Спб.: Наука, 1994. – 84 с.

Поступила в редколлегию 02.03.2010 г.

УДК 519.86:347.464

Інтервальні оцінювання параметрів для визначення надійності програмного забезпечення / В.Ю. Дубницький, А.М. Кобилін, О.А. Кобилін // Біоніка інтелекту: наук.-техн. журнал. – 2010. – № 1 (72). – С. 43–49.

Показано ефективність застосування інтервальних обчислень для розв'язання задач, пов'язаних з тестуванням програм. Показано, що застосування зворотного польського запису дозволяє скоротити обсяг програми, отже, підвищити надійність.

Табл. 7. Іл. 3. Бібліогр.: 5 найм.

UDK 519.86:347.464

Interval evaluation of parameters to determine software reliability / V.Y. Dubnickiy, A.M. Kobylin, O.A. Kobylin // Bionics of Intelligence: Sci. Mag. – 2010. – № 1 (72). – P. 43–49.

Efficiency of interval calculations application for solving the tasks connected to the programs testing is shown. It is also proved that using the backwards polish notation allows to reduce the volume of the program, and so to improve its reliability.

Table 7. Fig. 3. Ref.: 5 items.