УДК 004.054; 004.415.52; 004.423 doi: 10.15421/321727

А.Л. Хижа, И.Г. Высокопоясный
Днипровский национальный университет имени Олеся Гончара

АВТОМАТИЧЕСКАЯ ПРОВЕРКА СЕМАНТИЧЕСКОЙ ПРАВИЛЬНОСТИ РЕШЕНИЙ ЗАДАЧ ПО ПРОГРАММИРОВАНИЮ

В данной работе описан подход к обучению программированию, основанный на формальных спецификациях, и представлена разработанная авторами онлайновая Интернет-система, осуществляющая автоматическую статическую верификацию программного кода.

Дана робота описує підхід до навчання програмуванню, що базується на формальних специфікаціях, та презентує розроблене авторами програмне забезпечення онлайнової Інтернет-системи, що здійснює автоматичну статичну верифікацію програмного коду.

In this paper, we describe an approach to teaching programming based on formal specifications and represent a component of Internet online system for automating static program verification developed by the authors.

Ключевые слова: обучение программированию, формальная спецификация программ, предикаты первого порядка, слабейшие предусловия, доказательство правильности программ.

Введение. Формулирование спецификации программного кода перед его разработкой делает разработку проще и позволяет формально доказать семантическую правильность полученной программы. Поэтому, при обучении программированию, студента важно научить писать код, основываясь на его формальной спецификации. В данной статье мы описываем подход к обучению программированию, основанный на формальных спецификациях и представляем разработанный нами компонент системы удалённого обучения, осуществляющий автоматическую статическую верификацию. Мы используем предикаты логики первого порядка и преобразователь предикатов WP для формальной спецификации и статической верификации. Для проверки правильности предикатов используется программное обеспечение Simplify. Пользователь получает спецификацию программы, пишет её и загружает на сервер, после чего получает подтверждение её семантической правильности или же набор диагностических сообщений об ошибках. Система рассчитана на использование при обучении формальным методам в программировании или в продвинутом курсе императивного программирования.

Книга Дэвида Гриса «Наука программирования» [1] рассказывает о том, что можно и нужно доказывать компьютерные программы, как математические теоремы.

Для этих целей используются формальные методы, построенные на основе логики Хоара [2]. Ключевую роль занимает так называемая «тройка Хоара»:

$$\{Q\}S\{R\}, \tag{1}$$

где Q – предикат предусловия, S – код программы, а R – предикат постусловия. Тройка Хоара является предикатом и читается как «программа S, начавшая выполнение в состоянии Q, обязательно завершит выполнение в состоянии R». Предикат

$$Q \Rightarrow WP(S,R) \tag{2}$$

эквивалентен (1), где $WP(S,R)$ – это предикат слабейшего предусловия (англ. «weakest precondition») [3]. Таким образом, семантическая правильность программы определяется через истинность (тавтологичность) предиката (2) [3]. Такое доказательство называется статической верификацией (в отличие от тестирования, которое, очевидно, является динамической верификацией).

К паре предикатов Q и R можно относиться как к спецификации программы S, что позволяет сформулировать задание на написание S в терминах Q и R. По получении такой спецификации, разработчик применяет подходящие стратегии построения кода и дополняет её дополнительными предикатами и функциями: в частности, предикатами инвариантов цикла и ограничивающими функциями цикла (вариантом) [1].

Представляется актуальным предусматривать в системах онлайнового обучения компоненту, выполняющую статическую верификацию программного кода, написанного студентом по формальной спецификации Q, R, P, t.

Известны системы, в которых формальные методы применяются для обучения программированию. Обычно сопутствующие статьи имеют заголовки вроде «Система для автоматической оценки заданий по программированию».

Для сравнения таких работ мы применяли следующие критерии:

1. Использует система статическую или динамическую верификацию?
2. Если используются формальные методы, то используется ли логика Флойда-Хоара в чистом виде, слабейшие предусловия или что-то ещё?
3. Присутствует ли автоматическое применение формальных методов или это обучающий курс без автоматизации?
4. Какова сложность задач, ставящихся перед студентом (в смысле допустимых программных конструкций)?
5. Насколько полезные диагностические сообщения производит система в ответ на ошибки студента?

За исчерпывающим обзором подобных систем мы отсылаем читателя к статье П. Ихантола и др. [7].

Тут же мы проведём анализ системы, более близкой по духу к нашим образовательным целям, описанной в статье Т. Куана и др. [8]. В этой системе авторы пользуются статической верификацией, основанной на логике Флойда-Хоара. Так как авторы не используют слабейшие предусловия, они не мо-

гут определить, где именно произошла ошибка верификации, и вынуждены прибегать к проверке моделей для поиска контрпримеров. Авторы не предоставляют студенту формальную спецификацию программы в явном виде (она скрыта внутри системы). Возможно, именно поэтому описанная система работает только с заданиями вида «найти большее из двух чисел» – в статье явно сказано о том, что система работает самое большее с ветвлениями (в дальнейшем планируется поддержка циклов).

Вопрос диагностики ошибок верификации заслуживает особого внимания. В работе М. Кристакис и др. [9] ошибки верификации разделяются на две категории: настоящие ошибки, ложные предупреждения и таймауты (доказательство утверждения длилось слишком долго, и было искусственно прервано). В отличие от этой работы, в нашей системе у пользователя есть доступ к формальной спецификации программы, что почти полностью сводит эту проблему на нет.

В нашей работе, сфокусированной на обучении студентов программированию, мы будем предполагать, что инварианты и варианты циклов заданы как часть расширенной спецификации программы и даны студенту как часть задания.

Постановка задачи: разработать веб-приложение для обучения студентов написанию программного кода на основе расширенной спецификации. Студент выбирает задание в браузере, получает страницу с описанием задания и его расширенной спецификацией, составляет программу на процедурном языке программирования и отправляет её на проверку. Серверная часть системы проводит статическую верификацию полученной программы и отправляет студенту её результаты.

Метод решения и анализ полученных результатов. В качестве пользовательского языка программирования системы мы используем псевдокод, минимально, но достаточно поддерживающий парадигму алгоритмизации и элементарные структуры данных, такие как константы, простые переменные и массивы.

Для разбора пользовательского программного кода и предикатов формальной спецификации мы генерируем соответствующие компиляторы с помощью системы ANTLR [4].

Для преобразования кода программы в предикаты (условия верификации) мы пользуемся преобразователем предикатов WP. Правила трансформации пустой команды (skip), команды завершения программы с ошибкой (abort), присваивания, последовательности, ветвления (IF) и цикла (DO) описаны в книге Д. Гриса [1].

Мы разработали серверный скрипт, который использует описанные выше компиляторы для трансляции программы и её спецификаций, после чего применяет правила преобразования WP, чтобы получить условия верификации.

Для доказательства правильности условий верификации, полученных с помощью преобразователя WP, мы используем SAT-солвер Simplify [5].

Использование Simplify. Simplify – программа, использующая вариацию метода резолюций для доказательства истинности (тавтологичности) заданных предикатов. Simplify, в отличие от других подобных средств, никогда не уходит в бесконечный цикл в процессе доказательства, просто считая все предикаты, доказательство которых заходит в тупик, ложными (не тавтологиями). Таким образом, существенная часть решения нашей задачи состоит в том, чтобы убедиться, что Simplify располагает всеми необходимыми «знаниями» для доказательства каждого предиката. В нашем случае, ситуации, в которых Simplify необходима была наша «помощь», можно разделить на две группы:

1) определение или описание некоторых свойств искусственно введённых функций и предикатов (например, функция суммы элементов массива или предикат «массив A есть перестановкой массива B»);

2) определение тех свойств операций, «знания» которых мы ожидали от Simplify, но по какой-то причине были разочарованы (например свойство коммутативности умножения), и «помощь» в выделении релевантных в процессе доказательства посылок из кванторов всеобщности и существования.

Если к ситуациям из первого пункта списка мы были готовы и заранее планировали их решения, то проблемы из второго пункта, часто были неожиданностью. Далее мы по-очереди опишем ситуации из обоих пунктов и приведём некоторые примеры.

Определение новых функций и предикатов. Самой ожидаемой и простой для решения оказалась ситуация использования функции или предиката, неизвестных Simplify. Как пример, можно привести доказательство программы, вычисляющей сумму элементов массива b. Её спецификация не может обойтись без квантора суммы. Например, инвариант цикла для этой задачи выглядит так:

$$0 \le i \le N \wedge n = N \wedge b = B \wedge s = \sum_{k=0}^{i-1} B[k].$$

Очевидно, что квантор суммы тут не может быть раскрыт в последовательные сложения, так как количество слагаемых переменно. В данном случае, сумму можно считать функцией от массива и двух его индексов $\sum_{k=start}^{end} b[k] = sum(b, start, end)$. Определить эту функцию можно рекурсивно таким образом:

$$sum(b, start, end) = 0, \quad start > from.$$

$$sum(b, start, end) = sum(b, start, end-1) + b[end], \quad start \le from.$$

Далее, нам остаётся только «перевести» это выражение на язык Simplify:

```
(BG_PUSH
  (FORALL (b i j) (PATS (sum b i j))
    (AND
      (IMPLIES (> i j) (EQ (sum b i j) 0))
      (IMPLIES
        (<= i j)
        (EQ
          (sum b i j)
          (+ (select b j) (sum b i (- j 1))) )))))
```

Такого рода определения в нашей системе называются аксиомами. Аксиомы создаются пользователем (преподавателем) в отдельном файле и помещаются в соответствующую директорию *axioms*. Если задача требует какой-либо аксиомы, её название перечисляется в поле-списке *axioms* описания задачи.

Аксиомам не обязательно содержать полное определение предиката или функции. Например, аксиома для предиката «массив A есть перестановкой массива B», который используется в задаче сортировки массива, содержит только два свойства этого предиката, которых достаточно для доказательства генерируемых условий верификации:

```
(DEFPRED (perm a b n))

(BG_PUSH
  (FORALL (a b n i j)
    (IMPLIES
      (AND
        (perm a b n)
        (<= 0 j) (< j n)
        (<= 0 i) (< i n))
      (perm (store (store a j (select a i)) i
                                (select a j)) b n))))

(BG_PUSH
  (FORALL (a b n)
    (IMPLIES
      (FORALL (i)
        (IMPLIES
          (AND (<= 0 i) (< i n))
          (EQ (select a i) (select b i))))
      (perm a b n))))
```

Первое свойство означает, что от перестановки двух элементов массива **a** местами он не перестаёт быть перестановкой массива **b**, а второе утверждает, что два поэлементно равных массива являются перестановками друг друга. Причина использования этой аксиомы вместо полного определения **perm** через кванторы количества:

$$\left(\forall_i : 0 \le i < n : \left(N_k : 0 \le k < n : a[i] = a[k]\right) = \left(N_k : 0 \le k < n : a[i] = b[k]\right)\right)$$

кроется во внутреннем устройстве Simplify. Из полного определения пруверу довольно сложно вывести закономерности, необходимые для доказательства условий верификации, поэтому эту роль нам приходится брать на себя.

По похожим причинам, на самом деле, существует две аксиомы, задающие функцию суммы элементов массива: одна (приведенная выше) суммирует элементы с конца массива к началу, а вторая – с начала к концу. Вторая аксиома применяется в задаче вычисления суммы элементов массива, при условии движения по массиву с конца к началу. Необходимость двух аксиом связана с доказательством выражений вида

$$\left(s = \sum_{k=0}^{i-1} a[k]\right) \Rightarrow \left(s + a[i] = \sum_{k=0}^{i} a[k]\right)$$

и

$$\left(s = \sum_{k=i+1}^{n-1} a[k]\right) \Rightarrow \left(s + a[i] = \sum_{k=i}^{n-1} a[k]\right).$$

При использовании первой аксиомы первый предикат Simplify докажет, один раз «раскрыв» квантор суммы по определению:

$$\left(s = \sum_{k=0}^{i-1} a[k]\right) \Rightarrow \left(s + a[i] = \sum_{k=0}^{i} a[k]\right)$$

$$\left(s = \sum_{k=0}^{i-1} a[k]\right) \Rightarrow \left(s + a[i] = \sum_{k=0}^{i-1} a[k] + a[i]\right)$$

$$\left(s = \sum_{k=0}^{i-1} a[k]\right) \Rightarrow \left(s = \sum_{k=0}^{i-1} a[k]\right)$$

$$T$$

Второй же предикат доказать не удастся, потому что определение квантора суммы позволяет выносить наружу только последнее его слагаемое, а не первое, как требуется в доказательстве. При применении второй аксиомы ситуация обратная. Таким образом, обе аксиомы нужны, но для разных задач.

Проблемы реализованной системы задания аксиом. В нашей реализации работы с аксиомами на данный момент есть несколько проблем. Мы не можем определять аксиомы в достаточно общих терминах, так как Simplify не поддерживает переменные функции и предикаты. Например, мы не можем использовать аксиому суммы для определения скалярного произведения векторов (что нужно для задачи умножения матриц), так как суммирование происходит не по элементам одного массива, а по произведениям соответствующих элементов столбца и строки двойных массивов. Мы не можем определить квантор суммы как функцию высшего порядка, зависящую от некоторой другой функции $f(k)$, определяющей

слагаемое для каждого индекса k, поэтому в текущей реализации системы нам приходится определять разные аксиомы для этих двух случаев (а также для случая скалярного произведения строки матрицы и вектора, представленного одномерным массивом). Побочным эффектом такого низкого уровня возможности повторного использования аксиом есть необходимость изменять некоторые элементы кода системы после создания новой аксиомы. Так как разные контексты применения квантора суммы требуют использования разных аксиом, код, занимающийся переводом предикатов из внутреннего представления системы, в строку с предикатом в формате Simplify, должен определять структуру тела квантора суммы и, в зависимости от неё, выбирать нужную аксиому (либо приходить к выводу, что подходящей аксиомы нет среди указанных в описании задачи).

Этот аспект работы с системой довольно неудобен, и мы рассматриваем возможность его улучшения. Например, похоже, что квантор сложения (и, по аналогии, произведения и количества) произвольных выражений вида $\sum_{k=start}^{end} f(k)$ можно определить единственный раз для одномерного массива как $\sum_{k=start}^{end} \hat{a}[k]$, если убедиться, что имя мас сива $\hat{a}$ уникально в контексте рассматриваемого предиката и среди посылок, которыми в процессе доказательства будет располагать Simplify, присутствует такое определение элементов этого массива $\left(\forall_k : start \le k < end : \hat{a}[k] = f(k)\right)$.

Также, если код для «распознавания» подходящего контекста применения аксиомы всё-ещё будет необходим, его написание следует возложить на пользователя (преподавателя), определяющего эту аксиому. Таким образом, задание новой аксиомы потребует не только создания файла с Simplify кодом, но и, возможно, отдельного JavaScript файла, содержащего соответствующую функцию распознавания. При таких условиях внутренний код системы обязан будет проверять наличие определённой пользователем функции и обеспечивать её своевременный вызов, но уже не потребует изменения при создании новых аксиом.

Проблемы неполноты набора аксиом Simplify. Доказательство некоторых задач заканчивалось безуспешно по причинам недостаточного стандартного набора аксиом Simplify. Например, при проверке работы задания на вычисление произведения элементов массива автоматическое доказательство условий верификации не приводило к желаемому результату. И это при его полной аналогии с вычислением суммы, успешно протестированным ранее! После (относительно долгого) поиска причин такого поведения Simplify, выяснилось, что произведение двух целых чисел в Simplify не отвечает свойству коммутативности. Подтверждающий это вывод из интерактивного режима прувера:

```
>       (EQ (* a b) (* b a))
Counterexample:
  context:
    (AND
      (NEQ (* a b) (* b a))
    )
1: Invalid.
```

После определения аксиомы коммутативности:

```
(BG_PUSH
  (FORALL (a b c)
    (EQ
      (* a (+ b c))
      (+ (* a b) (* a c)) )))
```

задача вычисления произведения элементов массива была успешно доказана. Впоследствии эта аксиома повторно использовалась в нескольких других задачах.

Далее выяснилось, что в стандартном наборе аксиом Simplify также отсутствует аксиома ассоциативности умножения и дистрибутивности умножения относительно сложения, но, пока что, нам не понадобилось собственноручно доопределять эти аксиомы. Также проблемы возникали со стандартным набором аксиом теории отображений, на которой строится работа с массивами. Например, прувер не считает два поэлементно равных массива равными, если не ввести аксиому:

```
(BG_PUSH
  (IMPLIES
    (FORALL (k)
      (EQ (select a k) (select b k)))
    (EQ a b)))
```

Из этого, в частности, следует, что Simplify не способен вывести равенство $f(a) = f(b)$ для некой функции f, только на основе равенства двух массивов, а вынужден пытаться применить её определение, что в большинстве случаев не заканчивается успехом из-за переменной длинны массивов. Приведённой выше аксиомой это можно исправить, но в более сложном случае ограничиться одной аксиомой не выйдет. Например, если говорить о попытке вывести из поэлементного равенства срезов массивов равенство функций, зависящих от этих срезов. Для такой задачи потребуется новая аксиома для каждой функции. В задачах, присутствующих в нашей системе, подобных проблем пока не возникало, но мы считаем знание этих особенностей полезным для будущих пользователей системы, составляющих свои собственные задачи.

Использование стандартных кванторов Simplify. Simplify позволяет работать с предикатами, содержащими кванторы всеобщности и

существования. В общем виде квантор в формате Simplify можно записать так:

```
(FORALL\EXISTS (<связанные-переменные>)
               [(PATS <паттерны>)] <тело-квантора>)
```

В процессе доказательства квантор, стоящий в посылке, можно представить как бесконечный набор конъюнктов (в случае квантора всеобщности) или дизъюнктов (в случае квантора существования). Учесть в доказательстве весь этот набор, разумеется, невозможно, поэтому прувер пользуется только теми его частями, которые посчитает нужными. Для того, чтобы выделить конъюнкты или дизъюнкты, релевантные для доказательства следствий, Simplify генерирует для каждого квантора набор паттернов – термов, в которые входит каждая из связанных переменных. Каждый терм, встреченный при доказательстве, сравнивается с известными паттернами кванторов. Будем говорить, что терм t подходит под паттерн p, если существует такая замена связанных переменных в p на некоторые термы, результат которой равен t. Тогда, если в процессе доказательства один из термов t подходит под некий паттерн p, то из тела, соответствующего p квантора, путём применения той же самой замены, с помощью которой удалось привести p к t, формируется дополнительная посылка (гипотеза доказательства). При этом, если рассматривался квантор всеобщности, доказательство продолжается дальше обычным ходом, а если квантор существования, то, по аналогии с выбором одного из нескольких дизъюнктов, отделяется ветвь доказательства, в которой добавлена новая посылка, а использованный квантор недоступен. Если доказать предикат в этой ветви не удаётся, Simplify возвращается обратно к моменту создания новой посылки и продолжает доказательство, игнорируя её. Таким образом квантор существования становится доступен для использования в других контекстах.

Для случаев, когда автоматически построенных шаблонов оказывается недостаточно, Simplify предоставляет возможность определять пользовательские шаблоны в списке PATS (см. общий вид квантора). Нам пришлось воспользоваться этим механизмом при работе с вложенными кванторами, возникающими, например, в задачах с двумерными массивами. Подробного анализа проведено не было, но похоже на то, что Simplify просто игнорирует вложенные кванторы при поиске термов, подходящих на роль паттерна. Одним из возможных решений мог бы быть уход от вложенных кванторов в пользу кванторов по нескольким переменным одновременно. Для этого потребовалось бы обеспечивать отсутствие совпадений имён связанных переменных внутренних кванторов и свободных переменных тела внешнего квантора. В текущей реализации система решает эту проблему другим путём, а именно добавлением пользовательских паттернов, полученных из тела внутреннего квантора. Более конкретно, система добавляет в список паттернов первое из найденных выражений взятия элемента массива по индексу, где индекс зависит от переменной, связанной квантором.

Ещё одна проблема возникла при добавлении задачи поиска минимального (максимального) элемента массива. Её удобно проиллюстрировать на примере предиката $\exists k\ a[k] = a[0]$. Очевидно, что тело квантора обращается в истину при подстановке нуля вместо связанной переменной, но Simplify никогда не выберет нужный дизъюнкт, так как в окружающем квантор выражении нет ни одного терма, который бы подходил под сгенерированные паттерны. В задаче поиска минимума, выражение этого вида возникает в первом пункте доказательства правильности цикла:

$$0 < N \Rightarrow 1 \le 1 \le N \wedge (\forall k : 0 \le k < 1 : a[k] \le a[0]) \wedge (\exists k : 0 \le k < 1 : a[k] = a[0])$$

Счётчик цикла тут инициализируется единицей, а за минимальный элемент принимается первый элемент массива. Так как предикат не содержит термов, подходящих под паттерны квантора существования (какими бы они ни были), доказательство заканчивается неудачей. Решением, в нашем случае, послужило переопределение этого специфичного квантора существования, который невозможно было доказать обычным образом, с помощью аксиомы:

```
(DEFPRED (contains a b arr m))
(BG_PUSH
  (FORALL (a b arr m) (PATS (contains a b arr m))
    (AND
      (IMPLIES
        (> a b)
        (IFF (contains a b arr m) FALSE) )
      (IMPLIES
        (<= a b)
        (IFF
          (contains a b arr m)
          (OR
            (contains a (- b 1) arr m)
            (EQ (select arr b) m) ))))))
```

Этот подход работает, потому что теперь, встречая предикат $contains(0, 0, b, b[0])$, прувер не ищет термы, подходящие под один из сгенерированных паттернов, а сразу же раскрывает его по определению, получая $contains(0, -1, b, b[0]) \vee b[0] = b[0]$, что, очевидно, даёт истину.

Алгоритм работы системы.

1. Предоставить пользователю возможность выбрать задание.
2. Показать пользователю расширенную спецификацию для выбранной задачи.
3. Загрузить программный код пользователя.
4. Скомпилировать код программы. В случае обнаружения синтаксических ошибок, вернуть диагностические сообщения и перейти к шагу 2.
5. Преобразовать программу в набор условий верификации.

6. Проверить каждое из условий верификации, используя прувер Simplify.

7. Сообщить пользователю о результатах верификации.

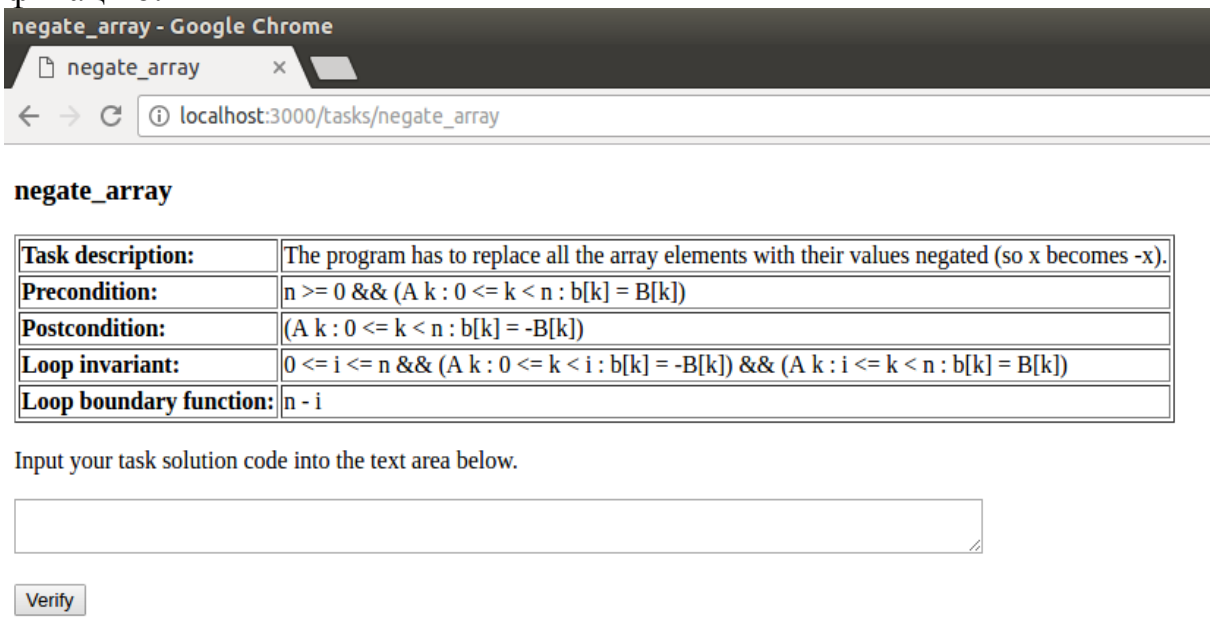
8. Вернуться на шаг 2.

Алгоритм описан.

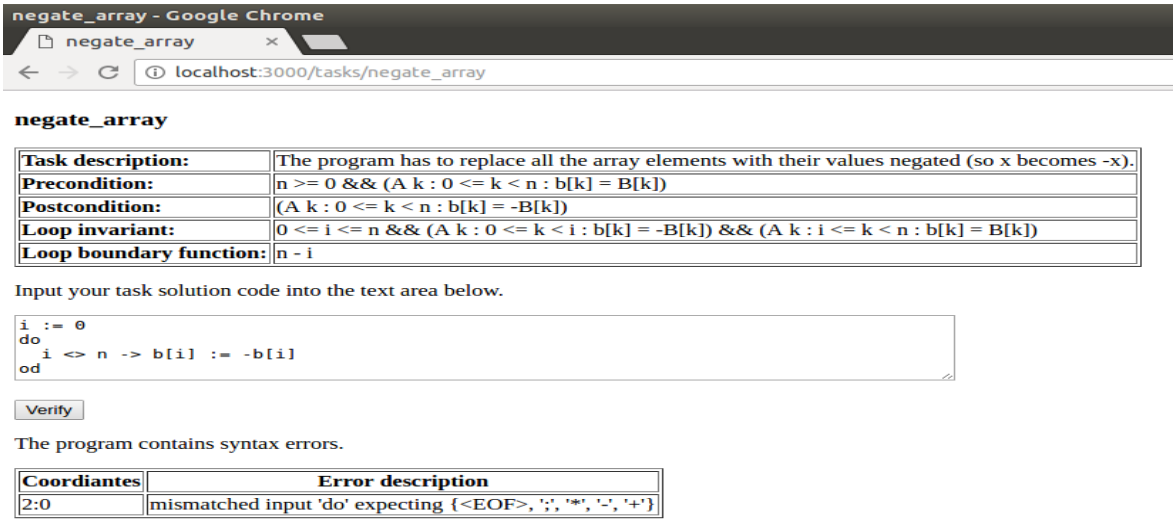
Пример работы системы. Работать с системой можно с помощью обычного веб-браузера. Все взаимодействия пользователя с системой происходят через веб-формы.

Например, пусть студенту задали решить задачу, пользуясь нашей системой. Для начала, студенту будет показан список задач, который выглядит так:

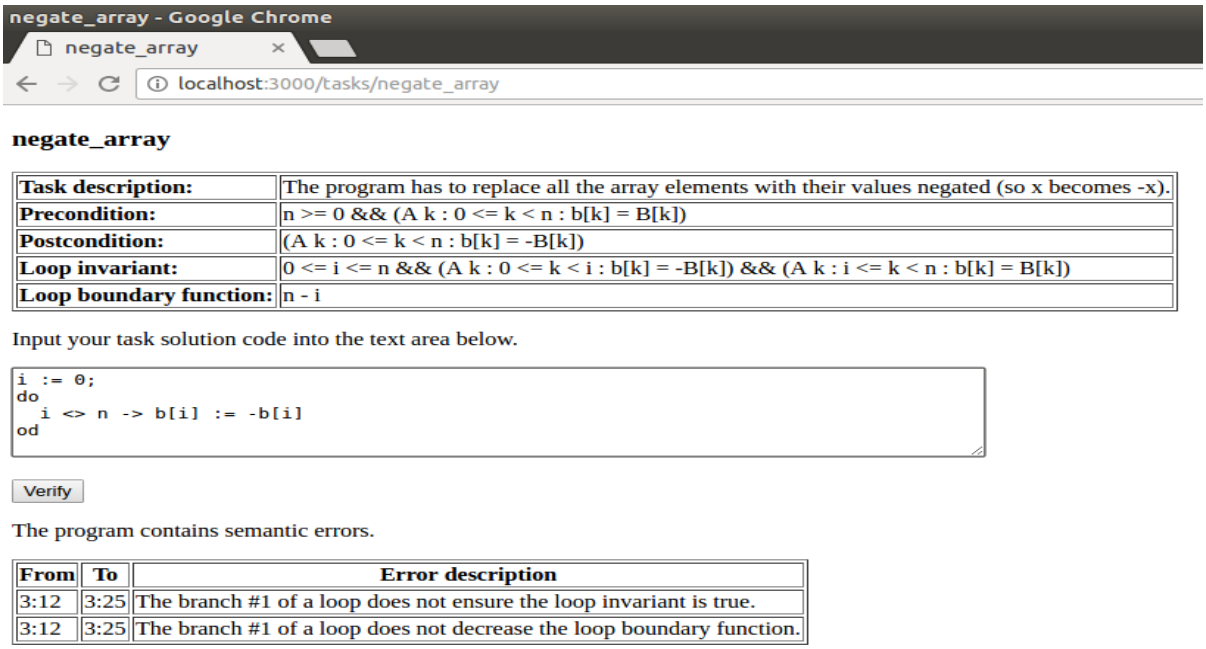
Студент выбирает задачу, которую ему задали решить, и видит её спецификацию:



Пусть студент написал некоторый программный код и нажал кнопку «Verify»:

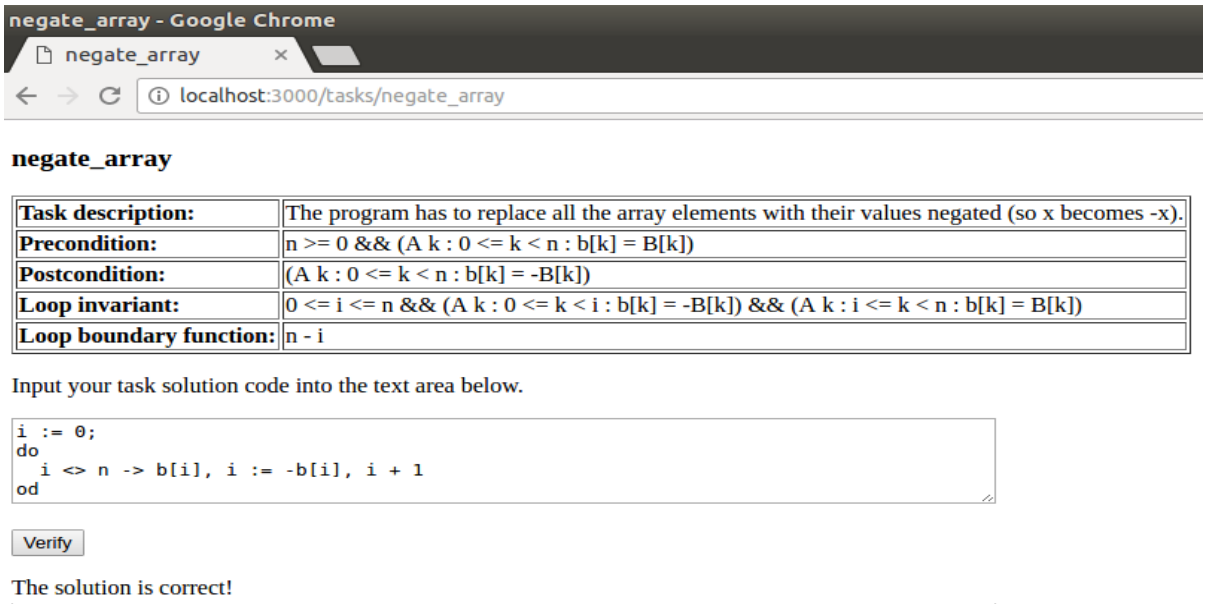


Похоже, что в программе присутствует синтаксическая ошибка. Пользуясь сообщением об ошибке, студент находит ошибку – недостающую точку с запятой – и пытается верифицировать исправленную версию программы.



Теперь похоже, что программа не отвечает заданной спецификации. После анализа сообщений об ошибках, студент понимает, что он забыл увеличить счётчик цикла. Из-за этого единственная ветка цикла не уменьшает ограничивающую функцию и пытается повторно сменить знак элементу массива, которому уже сменили знак, из-за чего не соблюдается инвариант цикла. Очевидно, что эта программа неверна, поэтому студент изменяет код и пытается верифицировать его ещё раз.

На этот раз верификация проходит без каких-либо ошибок, что значит, что теорема о правильности данной программы была успешно доказана.



Выводы. В данной статье описано веб-приложение для дистанционного обучения программированию. Его можно использовать для обучения студентов написанию программ, основываясь на расширенной формальной спецификации. Программы могут содержать последовательности команд, ветвления и циклы различной степени вложенности. Продемонстрированный подход можно обобщить для применения к программам**,** содержащим несколько циклов. Множество доступных пользователю языков программирования может быть расширено при условии, что новый язык можно отобразить в псевдокод, определённый в [1].

Библиографические ссылки

1. **Gries, D**. The Science of Programming [Text] / D. Gries. – Schpringer-Verlag, 1981. – 366 p.
2. **Hoare, C.A.R.** An axiomatic basis for computer programming [Text] / C.A.R. Hoare // Communications of the ACM, 12 (10):576–580 and 583, Oct. 1969.
3. **Dijkstra, E. W.** Guarded commands, nondeterminacy and formal derivation of programs [Text] / E. W. Dijkstra // Commun. ACM, 18:453–457, August 1975. ISSN 0001-0782. doi:10.1145/360933.360975.
4. The ANTLR Parser Generator – http://www.antlr.org/ – An electronic resource.
5. **Detlefs D**. Simplify: A Theorem Prover for Program Checking [Text] / D. Detlefs, G. Nelson, J. B. Saxe. **–** Hewlett-Packard Company, 2003. – 122 p.
6. **Pierce B.** Types and Programming Languages [Text] / B. Pierce. -MIT Press, 2002. - 645 p.
7. **Ihantola, P.** Review of recent systems for automatic assessment of programming assignments [Text] / P. Ihantola, T. Ahoniemi, V. Karavirta, O. Seppala. **–** In Proceedings of the 10th Koli calling international conference on computing education research (pp. 86-93). ACM.
8. **Quan, T. T.** A framework for automatic verification of programing exercises [Text] / T. T. Quan, P. H. Nguyen, T. H. Bui, L. V. Huynh, A. T Do // Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on (pp. 41-45). IEEE.
9. **Christakis, M.** Integrated Environment for Diagnosing Verification Errors [Text] / M. Christakis,; K. Leino, M. Rustan, P. Müller, V. Wüstholz,. In: TACAS. LNCS. Springer, 2016.

Надійшла до редколегії 14.05. 2017