УДК 004.7 DOI: 10.26102/2310-6018/2020.29.1.005

Синтаксис и операционная семантика целевого языка в реализации технологии «предположи и допусти» при объединении циклов для верификации программ

Д.В. Лысов
Воронежский государственный технический университет
Воронеж, Российская Федерация

Резюме: Loop Fusion – преобразование программы для объединения нескольких последовательных петель в одну – было изучено в основном для оптимизации компилятора. В работе предлагается новая стратегия объединения циклов, которая может объединить любые петли, даже петли с зависимостью данных. Показано, что это полезно для программы проверки, потому что может упростить инварианты цикла. Суть цикла слияния заключается в следующем: если состояние после первого цикла было известно, два тела цикла могут быть вычислены одновременно, независимо от данных путем переименования переменных программы. Loop Fusion создает программу, которая угадывает неизвестное состояние после первого цикла, недетерминированно выполняет слитый цикл, в котором переменные переименовываются, сравнивает угаданное состояние и состояние, фактически вычисленное слитой петлей, и, если они не совпадают, расходится. Последние два шага, сравнение и расхождение, имеют решающее значение для сохранения частичной корректности. Подход «предположи и допусти» назван так потому, что в дополнение к первому шагу (предположи), последние два шага могут быть выражены псевдоинструкцией «допусти», которая используется в проверке программы.

Ключевые слова: преобразование циклов, верификация программы, инварианты циклов, операционная семантика, целевой язык.

Для цитирования: Лысов Д.В. Синтаксис и операционная семантика целевого языка в реализации технологии «предположи и допусти» при объединении циклов для верификации программ. *Моделирование, оптимизация и информационные технологии»*. 2020;8(2). Доступно по: https://moit.vivt.ru/wp-content/uploads/2020/05/Lysov_2_20_1.pdf DOI: 10.26102/2310-6018/2020.29.1.005

The syntax and operational semantics of the target language in the implementing the «assume and allow» technology when combining loops for program verification

D.V. Lysov
Voronezh State Technical University, Voronezh, Russian Federation

Abstract: Loop Fusion – Converting a program to combine multiple consecutive loops into one – has been studied mainly for compiler optimization. The paper proposes a new strategy for combining loops, which can combine any loops even loops with data dependency. It is shown that this is useful for the verification program, because it can simplify cycle invariants. The essence of the merge cycle is as follows: if the state after the first cycle was known, two bodies of the cycle can be calculated simultaneously, regardless of the data by renaming the program variables. Loop Fusion creates a program that guesses an unknown state after the first cycle, non-deterministically executes a merged cycle in which variables are renamed, compares the guessed state and the state actually calculated by the merged loop, and, if they do not match, diverges. The last two steps of the comparison and the

discrepancy are crucial for maintaining partial correctness. The «suppose and allow» approach is so named because, in addition to the first step (suppose), the last two steps can be expressed by the «allow» pseudo-instruction used in program verification.

Keywords: cycle transformation, program verification, cycle invariants, operational semantics, target language.

For citation: Lysov D.V. The syntax and operational semantics of the target language in the implementing the «assume and allow» technology when combining loops for program verification. *Modeling, optimization and information technology*. 2020;8(2). Available by: https://moit.vivt.ru/wp-content/uploads/2020/05/Lysov_2_20_1.pdf DOI: 10.26102/2310-6018/2020.29.1.005 (In Russ).

Введение

Объединение циклов [1, 4] – это преобразование программы для слияния нескольких последовательных циклов в один. Например, следующий код C

```
for (i = 0; i < n; i++) { sum1 += i; }
for (i = 0; i < n; i++) { sum2 += i; }
```

преобразуется в

```
for (i = 0; i < n; i++) {
  sum1 += i;
  sum2 += i;
}
```

с помощью объединения циклов. Хотя объединение циклов изучалось в основном для оптимизации компилятора, в работе исследуется применение объединения циклов для статической верификации программ.

Объединение циклов может быть полезным, поскольку объединённый цикл может иметь более простое инвариантное условие цикла. Для того, чтобы обосновать объединение циклов для статической верификации, рассмотрим следующую функцию C, которая включает в себя два вышеприведённых цикла:

```
void sample_01(int n) {
  int sum1 = 0, sum2 = 0;
  int i;
  for (i = 0; i < n; i++) { sum1 += i; }
  for (i = 0; i < n; i++) { sum2 += i; }
  assert (sum1 == sum2);
}
```

Легко видеть, что утверждение в конце всегда верно для любого аргумента n, но на самом деле статическому верификатору программ это трудно доказать. Фактически, таким средствам верификации моделей программного обеспечения, как CPAchecker [2] и SeaHorn [6] не удалось верифицировать эту функцию за 900 с в экспериментальной среде. Возможно это произошло потому, что инварианты цикла для доказательства утверждения включают нелинейную арифметику: $\left(\text{sum1} = \frac{\text{i}(\text{i}-1)}{2}\right) \wedge (\text{i} \le \text{n}) \wedge (\text{sum2} = 0)$ для первого цикла и $\left(\text{sum1} = \frac{\text{n}(\text{n}-1)}{2}\right) \wedge (\text{i} \le \text{n}) \wedge (\text{sum2} = \frac{\text{i}(\text{i}-1)}{2})$ для второго цикла. С другой стороны, если заменить два цикла одним объединённым

```
void sample_01_fused(int n) {
  int sum1 = 0, sum2 = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum1 += i;
    sum2 += i;
  }
  assert (sum1 == sum2);
}
```

то станет ясно, что есть более простой инвариант цикла **sum1=sum2**, а CPAchecker и SeaHorn успешно доказывают отсутствие ошибки утверждения в течение нескольких секунд.

Тем не менее, применимость стандартных методов для объединения циклов довольно ограничена. Например, объединение циклов в [1] требует выполнения следующих двух условий для сохранения семантики программы:

1. Два цикла имеют одинаковые границы цикла.
2. Тело первого цикла не зависит от тела второго.

Таким образом, следующие два цикла, в которых одинаковая переменная **sum** обновляется, не удовлетворяют условию выше и поэтому не могут быть объединены.

```
for (i = 0; i < n; i++) { sum += i; }
for (i = 0; i < n; i++) { sum -= i; }
```

На самом деле, прямое применение объединения циклов к этим циклам сохраняет семантику, и средства верификации моделей могли бы верифицировать объединённый цикл. Конечно, в целом это не так.

1. Постановка задачи

Предлагается преобразование программы, которое решает эту проблему. Преобразование основано на следующем наблюдении: можно легко объединить *любые* два цикла, если в начале выполнения программы можно предсказать значения, сохраняемые переменными в тот момент, когда выполнение программы достигнет точки между циклами. Это наблюдение продемонстрировано с помощью пошаговых преобразований следующей программы **P**, которая включает в себя вышеприведенные циклы:

```
void sample_02(int n) {
  int sum = 0;
  int i;
  for (i = 0; i < n; i++) { sum += i; }
  for (i = 0; i < n; i++) { sum -= i; }
  assert(sum == 0);
}
```

Сначала попытаемся разрешить зависимость между телами двух циклов. Простой способ добиться этого – ввести новую переменную **sum2**, соответствующую переменной **sum**, и заменить вхождение **sum** во втором цикле на **sum2**.

```
int sum = 0; int sum2 = 0;
int i;
for (i = 0; i < n; i++) { sum += i; }
/* (X) */
for (i = 0; i < n; i++) { sum2 -= i; }
assert(sum == 0);
```

Хотя можно объединить два цикла в этой результирующей программе по стандартной методике, но это не соответствует задаче. Эта программа нарушает соотношение между первым циклом и вторым циклом в исходной программе и, таким образом, семантически отличается от **P**. Следовательно, верификация результирующей программы не подразумевает безопасность **P**.

Чтобы восстановить это соотношение между двумя циклами, вводится переменная **sum_middle**, чтобы представить значение **sum**, когда управление доходит до строки (**X**).

```
int sum = 0;
int sum2;
int i;
for (i = 0; i < n; i++) { sum += i; }
assume(sum_middle == sum);
sum2 = sum_middle;
/* (Y) */
for (i = 0; i < n; i++) { sum2 -= i; }
sum = sum2;
assert(sum == 0);
```

Назовем эту программу **Q**. Предположим, что можно каким-то образом предсказать значение **sum_middle** *до начала выполнения* **Q**. Тогда безопасность **Q** (отсутствие отказа **assert**) подразумевает безопасность **P**, потому что:

• значения **sum** и **sum2** в строке (**Y**) равны **sum_middle** (благодаря прогнозу);

• значение **sum2** копируется в **sum** перед выполнением **assert(sum==0)**.

Здесь оператор **assume(sum==sum_middle)** работает как холостая команда, если выполняется **sum=sum_middle**; он расходится, если **sum=sum_middle** не выполняется и поэтому выполнение не продолжается. Эта команда используется, например, в Boogie [11] для выражения допущений, используемых при верификации в качестве программы.

Последняя проблема заключается в том, как предсказать значение **sum_middle** в начале. Для верификации безопасности, которая является целью, оказалось, что достаточно *предположения* значения:

```
int sum_middle = havoc();
int sum = 0, sum2 = sum_middle, i;
for (i = 0; i < n; i++) { sum += i; }
for (i = 0; i < n; i++) { sum2 -= i; }
assume(sum == sum_middle);
sum = sum2;
assert(sum == 0);
```

Команда **havoc()** недетерминированно выбирает значение и возвращает его. Опущен ниже оператор **assume**, который не использует **sum2**, поднят **sum2=sum_middle**, который не использует **sum**. Поскольку два цикла в программе

теперь удовлетворяют условиям объединения циклов, можно объединить эти циклы, чтобы получить следующую программу:

```
void sample_02_fused(int n) {
  int sum_middle = havoc();
  int sum = 0, sum2 = sum_middle, i;
  for (i = 0; i < n; i++) {
    sum += i; sum2 -= i;
  }
  assume(sum == sum_middle);
  sum = sum2;
  assert(sum == 0);
}
```

Результирующая программа сначала *предполагает* значение **sum** в строке (**X**) в **P**, выполняет программу, а затем *допускает*, что предположенное значение действительно является правильным предположением, используя **assume**, чтобы исключить выполнение, которое невозможно в **P**. Основным вкладом исследования является формализация объединения циклов с использованием метода *«предположения и допущения»*, доказательство его обоснованности при верификации безопасности и эмпирическая демонстрация его полезности.

При этом результирующая программа **sample_02_fused** *не совсем семантически эквивалентна* исходной программе **sample_02**: если предполагаемое значение **sum_middle** отличается от фактического значения **sum** в середине двух циклов исходной программы, то функция **sample_02_fused** расходится, тогда как **sample_02** всегда завершается. Это различие, однако, не является проблематичным в этом случае, поскольку основная цель – верификация безопасности, которая является частичной корректностью, и, следовательно, для преобразования требуется только эквивалентность до момента завершения.

2. Целевой язык

В этом разделе формализован синтаксис и операционная семантика целевого языка, который является вариантом языка IMP [10]. Формализация в основном стандартная, за исключением того, переменные, используемые в программах, сделаны более явными в процессе формализации, чтобы упростить подтверждение корректности объединения циклов.

2.1. Синтаксис

Фиксируется множество n-арных операций OP_n для каждого неотрицательного целого числа n (если n=0, они рассматриваются как постоянные значения). Для каждого конечного множества переменных V множество Exp_V выражения e_V и множество Com_V команд P_V определяются следующим образом:

Определение 2.1 (Выражения и Команды)

$$e_V \in Exp_V \triangleq x \mid f_n \underbrace{(e_V, ..., e_V)}_{n}$$

$$P_V \in Com_V \triangleq \begin{array}{|l} skip \mid x := e_V \mid P_V ; P_V \\ \text{if } e_V \text{ then } P_V \text{ else } P_V \\ \text{while } e_V \text{ do } P_V \text{ done} \\ \text{havoc } x \mid \text{assume } e \end{array}$$

x и f_n находятся в пределах V и Op_n, соответственно.

Используются целочисленные выражения для условий операторов **if** и **while** и рассматривается любое ненулевое значение как истинное. **Havoc** x – это команда, которая недетерминированно присваивает случайное целое число переменной x. **Assume** e допускает дополнительное условие, состоящее в том, что e оценивается как ненулевое значение. В практическом плане, если e оценивается как истинное, то имеется холостая команду; в противном случае она расходится. Остальные конструкции такие же, как и обычный язык **while**. Допускаются обычные операции, такие как $\neg \in Op_1$, $= \in Op_2$ и $\vee \in Op_2$ (со стандартной семантикой) и используем = и $\vee$ в качестве инфиксных операторов.

Можно было бы обойтись без **assume** e, поскольку оно может быть выражено комбинацией других команд, но эта команда включена для упрощения записи. Команда **assert**, используемая в примерах, не включена, поскольку она не нужна для технической разработки.

2.2. Операционная семантика

Семантика задается с использованием отношения $P, \sigma_1 \Downarrow \sigma_2$, читаемого как «выполнение команды P в состоянии σ_1 заканчивается в конечном состоянии σ_2», где P – команда, а σ_1 и σ_2 – состояния. Предположим, что каждой n-арной операции f_n ставится в соответствие отображение из $\mathbf{Z}^n$ в **Z** (также обозначаемое как f_n).

Определение 2.2. (Состояния). Состояние $\sigma \in State_V$ – это отображение из V в **Z**. Запишем $\sigma \mid W$ для ограничения σ на области W.

Определение 2.3. (Семантика выражений). Семантика $\sigma[\![e]\!]$ выражения e в состоянии σ определяется следующим образом:

$$\sigma[\![x]\!] \triangleq \sigma(x)$$
$$\sigma[\![f_n(e_1, ..., e_n)]\!] \triangleq f_n(\sigma[\![e_1]\!], ..., \sigma[\![e_n]\!])$$

Определение 2.4. (Обновление состояния). Для состояния $\sigma \in State_V$, $x \in V$ и $a \in \mathbf{Z}$, состояние $\sigma[x \mapsto a] \in State_V$ определяется следующим образом:

$$\sigma[x \mapsto a](y) \triangleq \begin{cases} a, x = y \\ \sigma(y), x \neq y \end{cases}$$

Определение 2.5. (Операционная семантика). Для состояний $\sigma_1, \sigma_2 \in State_V$ и команды $P \in Com_V$ соотношение $P, \sigma_1 \Downarrow \sigma_2$ определяется правилами на Рисунке 1.

Правила стандартные. В E_HAVOC целое число n выбрано недетерминировано. Правило E_ASSUME означает, что **assume** e – холостая команда, если e истинно (т.е. не

ноль). В противном случае она расходится. Таким образом, для случая $\sigma[\![e]\!]=0$ не существует правила.

$\boxed{P,\ \sigma_1 \Downarrow \sigma_2}$

$$\frac{}{\text{skip},\ \sigma \Downarrow \sigma}\ \text{E\_Skip} \qquad \frac{}{x{:=}e,\ \sigma \Downarrow \sigma[x \mapsto \sigma[\![e]\!]]}\ \text{E\_Assign} \qquad \frac{P_1,\ \sigma_1 \Downarrow \sigma_2 \quad P_2,\ \sigma_2 \Downarrow \sigma_3}{P_1;\ P_2,\ \sigma_1 \Downarrow \sigma_3}\ \text{E\_Seq}$$

$$\frac{\sigma[\![e]\!]\neq 0 \quad P_1,\ \sigma_1 \Downarrow \sigma_2}{\text{if } e \text{ then } P_1 \text{ else } P_2,\ \sigma_1 \Downarrow \sigma_2}\ \text{E\_IfT} \qquad \frac{\sigma[\![e]\!]=0 \quad P_2,\ \sigma_1 \Downarrow \sigma_2}{\text{if } e \text{ then } P_1 \text{ else } P_2,\ \sigma_1 \Downarrow \sigma_2}\ \text{E\_IfF}$$

$$\frac{\sigma[\![e]\!]=0}{\text{while } e \text{ do } P \text{ done},\ \sigma \Downarrow \sigma}\ \text{E\_WhileEnd} \qquad \frac{\sigma[\![e]\!]\neq 0 \quad P,\ \sigma_1 \Downarrow \sigma_2 \quad \text{while } e \text{ do } P \text{ done},\ \sigma_2 \Downarrow \sigma_3}{\text{while } e \text{ do } P \text{ done},\ \sigma_1 \Downarrow \sigma_3}\ \text{E\_WhileLoop}$$

$$\frac{n \in \mathbb{Z}}{\text{havoc } x,\ \sigma \Downarrow \sigma[x \mapsto n]}\ \text{E\_Havoc} \qquad \frac{\sigma[\![e]\!]\neq 0}{\text{assume } e,\ \sigma \Downarrow \sigma}\ \text{E\_Assume}$$

Рисунок 1 – Операционная семантика
Figure 1 – Operational semantics

2.3. Утверждения и тройки Хоара

Определим утверждения и тройки Хоара целевого языка. Для простоты утверждения семантически определены: для них не фиксируем синтаксис, а утверждение дается в виде множества состояний.

Определение 2.6. (Утверждения). Утверждение, ограниченное φ и ψ, в V является подмножеством State_V. Записываем Assertion_V для множества утверждений в V. Другими словами, множество Assertion_V является множеством всех подмножеств State_V.

Расширим ограничения на утверждения следующим образом: для таких множеств переменных V и W, при которых $V \subseteq W$ и утверждение $\varphi \in \text{Assertion}_W$, определим утверждение $\varphi|V \in \text{Assertion}_V$ как $\{\sigma_2 | \sigma_1 \in \varphi \wedge \sigma_1|v = \sigma_2\}$.

Определение 2.7. (Тройки Хоара). Для команды $P \in \text{Com}_V$ и двух утверждений φ и ψ в Assertion_V запишем: $\{\varphi\}P\{\psi\}$ тогда и только тогда, когда выполняется следующее условие:

$$\forall \sigma_1, \sigma_2 \in \text{State}_V : (\sigma_1 \in \varphi) \wedge (P, \sigma_1 \Downarrow \sigma_2) \Rightarrow \sigma_2 \in \psi$$

Для удобства обозначений введем разделяющую конъюнкцию [12]. Это не синтаксический конструктор, а оператор двух утверждений.

Определение 2.8. (Разделяющая конъюнкция). Пусть V и W – непересекающиеся множества переменных. Пусть σ_1 и σ_2 – состояния в State_V, State_W соответственно. Определим $\sigma_1 \cup \sigma_2 \in \text{State}_{V \cup W}$ следующим образом:

$$(\sigma_1 \cup \sigma_2) = \begin{cases} \sigma_1(x), x \in V \\ \sigma_2(x), x \in W \end{cases}$$

Определим *разделяющую конъюнкцию* $\varphi^*\psi$ двух утверждений $\varphi \in \text{Assertion}_V$ и $\varphi \in \text{Assertion}_W$:

$$\varphi^*\psi = \{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in \varphi, \sigma_2 \in \psi\}$$

2.4. V-эквивалентность

В этом разделе вводится понятие V-эквивалентности для выражения корректности объединения циклов. Обычно эквивалентность между P_1 и P_2 определяется следующим: для любой пары состояний σ_1 и σ_2, $P_1,\sigma_1 \Downarrow \sigma_2$ тогда и только тогда, когда $P_2,\sigma_1 \Downarrow \sigma_2$. Однако, нельзя ожидать, что объединение циклов даст эквивалентную команду в этом смысле, поскольку оно вводит недетерминированно расходящееся поведение и дополнительные переменные. Можно ожидать, что команда после объединения циклов покажет то же поведение ввода-вывода, что и раньше – *только когда она завершится в какой-то момент времени и только для переменных в исходной команде.* Таким образом, индексируем эквивалентность множеством переменных V, соответствующих переменным в исходной команде, и рассматриваем команды без сильной зависимости [3] V от других переменных.

Определение 2.9. (P зависит только от V). Для таких множеств переменных V и W, когда $V \subseteq W$ и $P \in Com_W$, будем говорить, что P *находится в зависимости только от* V тогда и только тогда, когда для любого $\sigma_1,\sigma_2 \in State_W$, такого, когда $\sigma_1|_V = \sigma_2|_V$, если $P,\sigma_1 \Downarrow \sigma'_1$ и $P,\sigma_2 \Downarrow \sigma'_2$, то $\sigma'_1|_V = \sigma'_2|_V$.

Как устанавливает следующее вспомогательное утверждение, для команды, зависящей только от V, можно ослабить предусловие для переменных, отличных от V, не влияя на постусловие для V.

Вспомогательное утверждение 2.10. *Пусть* $P \in Com_W$ *– команда, зависящая только от* V. $\{\varphi\}P\{\psi^*State_{W\setminus V}\}$ *тогда и только тогда, когда* $\{\varphi|_V^*State_{W\setminus V}\}P\{\psi^*State_{W\setminus V}\}$.

Определение 2.11. (V-эквивалентность). Пусть две команды $P_1 \in Com_{V1}$ и $P_2 \in Com_{V2}$ зависят только от V. P_1 и P_2 V-*эквивалентны*, записывается $P_1 \sim_V P_2$, тогда и только тогда, когда: для любых утверждений $\varphi,\psi \in Assertion_V$, $\{\varphi|^*State_{V1\setminus V}\}P_1\{\psi^*State_{V1\setminus V}\}$ тогда и только тогда, когда $\{\varphi|^*State_{V2\setminus V}\}P_2\{\psi^*State_{V2\setminus V}\}$.

Как показано ниже в утверждении 2.14, V-эквивалентность замкнута в контекстах, относящихся только к V. В следующем формально определим объединение циклов и докажем, что оно дает V-эквивалентную команду (где V – множество переменных в исходной команде). Утверждение гарантирует, что применение объединения циклов (к подкоманде) сохраняет частичную корректность.

Далее контекст C в $Context_V$ определяется как команда ($\in Com_V$), которая содержит поле [], а C[P] обозначает команду, полученную путем замены [] в C на P.

Вспомогательное утверждение 2.12. *Пусть* V, V_1 и V_2 *– такие множества переменных, что* $V \subset V_1$ *и* $V \subset V_2$, *а* $P_1,P_2 \in Com_{V1}$ *и* $P_3,P_4 \in Com_{V2}$. *Если* P_i *зависит только от V, при* $i \in \{1,...,4\}$ *и* $P_1 \sim_V P_3$, *а* $P_2 \sim_V P_4$, *тогда* $P_1;P_2 \sim_V P_3;P_4$.

Доказательство. Покажем, что из $\{\varphi^*State_{V1\setminus V}\}P_1;P_2\{\psi^*State_{V1\setminus V}\}$ следует $\{\varphi^*State_{V2\setminus V}\}P_3;P_4\{\psi^*State_{V2\setminus V}\}$. Пусть $\chi = \{\sigma' \mid \sigma \in \varphi State_{V1\setminus V} \wedge (P,\sigma \Downarrow \sigma')\}$. По определению $\{\varphi^*State_{V1\setminus V}\}P_1(\chi)$ и легко показать, что $\{\varphi^*State_{V1\setminus V}\}P_1\{\chi|_V^*State_{V1\setminus V}\}$. Аналогично, $\{\chi\}P_2\{\psi^*State_{V1\setminus V}\}$.

Поскольку P_1 зависит только от V, имеем $\{\chi|_V^*State_{V1\setminus V}\}P_2\{\psi^*State_{V1\setminus V}\}$ по вспомогательному утверждению 2.10. Имеем $\{\varphi^*State_{V2\setminus V}\}P_3\{\chi|_V^*State_{V2\setminus V}\}$ и $\{\chi|_V^*State_{V2\setminus V}\}P_4\{\psi^*State_{V2\setminus V}\}$, откуда $P_1 \sim_V P_3$ и $P_2 \sim_V P_4$ соответственно. Доказательство завершено.

Вспомогательное утверждение 2.13. *Для таких команд* $P_1 \in Com_{V1}$ *и* $P_2 \in Com_{V2}$, *при которых* $P_1 \sim_V P_2$ *и* $e \in Exp_V$,

while e **do** P_1 **done** ~$_V$ **while** e **do** P_2 **done**.

Доказательство. Для одного направления покажем, что

- если {φ*$State_{V1\setminus V}$} **while** e **do** P_1 **done** {ψ*$State_{V1\setminus V}$},
- ∀ σ∈{φ*$State_{V2\setminus V}$}, **while** e **do** P_2 **done**, $\sigma \Downarrow \sigma'$,
- тогда σ’∈{ψ*$State_{V2\setminus V}$}.

Это видно по индукции на выводе **while** e **do** P_2 **done**, $\sigma \Downarrow \sigma'$. Другое направление аналогично.

Утверждение 2.14. *Для любого* C *в* $Context_V$ *и команд* $P_1 \in Com_{V1}$ *и* $P_2 \in Com_{V2}$, *при которых* $P_1 \sim_V P_2$, выполнено $C[P_1] \sim_V C[P_2]$.

Доказательство. По индукции на структуре *C*. Для последовательной композиции P_1,P_2 используем вспомогательное утверждение 2.12, а для **while** – вспомогательное утверждение 2.13.

Утверждение доказано.

3. Объединение циклов

В этом разделе определим преобразование объединения циклов и покажем, что оно выдает V-эквивалентную команду.

Начнем с нескольких предварительных определений.

Определение 3.1. (Штриховые (прим) обозначения). Для множества переменных $V=\{x_1,\ldots,x_n\}$ определим $V'=\{x'_1,\ldots,x'_n\}$ как множество переменных, непересекающееся с V и имеющее тот же размер, что и V. При заданной команде $P \in Com_V$ определим P’ для команды (в $Com_{V'}$), которая задается заменой каждой переменной x_i в P на x'_i в V’.

Определение 3.2. Для множества переменных $V=\{x_1,\ldots,x_n\}$ определим $\vec{V} := V'$, как аббревиатуру команды $x_1 := x'_1; \ldots; x_n := x'_n$, и записываем **assume** $\vec{V} := V'$ как аббревиатуру команды **assume** $x_1 := x'_1; \ldots;$ **assume** $x_n := x'_n$.

3.1. Объединение двух последовательных циклов

Определение 3.3. (Объединение двух последовательных циклов). Пусть следующей командой будет $P \in Com_V$:

P=**while** e_1 **do** P_1 **done**; **while** e_2 **do** P_2 **done**.

Для такой команды P определим команду $Fusion[P] \in Com_{V \cup V' \cup V''}$, которая состоит из одного цикла **while**, следующим образом:

$$
\begin{aligned}
&\textsc{Fusion}[P] = \text{havoc } \vec{V}';\\
&\quad \vec{V}'' := \vec{V}';\\
&\quad \text{while } e_1 \vee e'_2 \text{ do}\\
&\quad\quad \text{if } e_1 \text{ then } P_1 \text{ else skip};\\
&\quad\quad \text{if } e'_2 \text{ then } P'_2 \text{ else skip}\\
&\quad \text{done};\\
&\quad \text{assume } \vec{V} = \vec{V}'';\\
&\quad \vec{V} := \vec{V}'
\end{aligned}
$$

Утверждение 3.4. *Для любого* $P \in Com_V$, Fusion[P] *не зависит от* V.

Утверждение 3.5. *Для любого* Fusion[P] P *и* Fusion[P] V-*эквивалентны*.

Доказательство (по рис. 2). Определим эквивалентность $P_1 \sim P_2$ следующим образом: $P_1 \sim P_2$ тогда и только тогда, когда для любых утверждений φ, ψ выполняется:

{φ}P1{ψ} если {φ}P2{ψ}.

Нетрудно показать, что свойство эквивалентности ~ замкнуто в контекстах (независимо от переменных) и что если $P_1 \sim P_2$, а P_1 и P_2 зависят только от V, то тогда $P_1 \sim_V P_2$.

На каждом из шагов используем вспомогательные утверждения, чтобы показать V-эквивалентность или ~ между некоторыми объектами, а затем применяем утверждение 2.14 или свойства ~, указанные выше. Легко показать, что все промежуточные команды зависят только от V. Первый шаг, который показывается вспомогательным утверждением 3.6, переименовывает переменные во втором цикле. Второй шаг, показанный вспомогательным утверждением 3.7, заменяет присвоения у **havoc** и **assume**. Третий шаг, который показывается вспомогательным утверждением 3.8, вводит $\vec{V}''$, чтобы избежать использования $\vec{V}'$, обеспечивая четвертый шаг, когда происходит обмен операторами, которые используют непересекающиеся множества переменных и который показан вспомогательным утверждением 3.9. Наконец, объединяются два последовательных цикла, которые относятся к непересекающимся множествам переменных. Это показывает вспомогательное утверждение 3.12. Доказательство завершено.

$$\begin{array}{l}\text{while } e_1 \text{ do } P_1 \text{ done;}\\ \text{while } e_2 \text{ do } P_2 \text{ done}\end{array} \sim_V \begin{array}{l}\text{while } e_1 \text{ do } P_1 \text{ done;}\\ \vec{V}' := \vec{V};\\ \text{while } e_2' \text{ do } P_2' \text{ done;}\\ \vec{V} := \vec{V}'\end{array} \sim_V \begin{array}{l}\text{while } e_1 \text{ do } P_1 \text{ done;}\\ \text{havoc } \vec{V}'; \text{ assume } \vec{V} = \vec{V}';\\ \text{while } e_2' \text{ do } P_2' \text{ done;}\\ \vec{V} := \vec{V}'\end{array}$$

$$\sim_V \begin{array}{l}\text{while } e_1 \text{ do } P_1 \text{ done;}\\ \text{havoc } \vec{V}'; \vec{V}'' := \vec{V}';\\ \text{assume } \vec{V} = \vec{V}'';\\ \text{while } e_2' \text{ do } P_2' \text{ done;}\\ \vec{V} := \vec{V}'\end{array} \sim_V \begin{array}{l}\text{havoc } \vec{V}'; \vec{V}'' := \vec{V}';\\ \text{while } e_1 \text{ do } P_1 \text{ done;}\\ \text{while } e_2' \text{ do } P_2' \text{ done;}\\ \text{assume } \vec{V} = \vec{V}'';\\ \vec{V} := \vec{V}'\end{array} \sim_V \begin{array}{l}\text{havoc } \vec{V}'; \vec{V}'' := \vec{V}';\\ \text{while } e_1 \vee e_2' \text{ do}\\ \quad \text{if } e_1 \text{ then } P_1 \text{ else skip;}\\ \quad \text{if } e_2' \text{ then } P_2' \text{ else skip}\\ \text{done;}\\ \text{assume } \vec{V} = \vec{V}'';\\ \vec{V} := \vec{V}'\end{array}$$

Рисунок 2 – Доказательство утверждения 3.5
Figure 2 – Proof of statement 3.5

Вспомогательное утверждение 3.6. *Для любых* $e \in Exp_V$ *и* $P \in Com_V$

$$\text{while e do P done} \sim_V \begin{array}{l}\tilde{V}' := \tilde{V};\\ \text{while e' do P' done;}\\ \tilde{V} := \tilde{V}'\end{array}$$

Вспомогательное утверждение 3.7.

$$\tilde{V}' := \tilde{V} \sim \text{havoc } \tilde{V}'; \text{ assume } \tilde{V} := \tilde{V}'.$$

Вспомогательное утверждение 3.8.

$$\text{assume } \vec{V} = \vec{V}' \sim_V \begin{array}{l}\vec{V}'' := \vec{V}'\\ \text{assume } \vec{V} = \vec{V}''\end{array}$$

Вспомогательное утверждение 3.9. *Если* $P_i \in Com_{Vi}$ *при* $i \in \{1,2\}$ *и* $V_1 \cap V_2 = \varnothing$, *то* $P_1;P_2 \sim P_2;P_1$.

Вспомогательное утверждение 3.10. *При* $P \in Com_V$, $\sigma_1,\sigma_3 \in State_V$, $\sigma_2,\sigma_4 \in State_W$, *при котором* $V \cap W = \varnothing$, *если* $P,\sigma_1 \cup \sigma_2 \Downarrow \sigma_3 \cup \sigma_4$, *то* $\sigma_2=\sigma_4$ *и* $P,\sigma_1 \Downarrow \sigma_3$.

Вспомогательное утверждение 3.11. *Пусть* **while** e_i **do** p_i **done** $\in Com_{Vi}$ *и* $V_1 \cap V_2 = \varnothing$, *и пусть* P *будет*

while $e_1 \vee e_2$ **do**
  if e_1 **then** P_1 **else skip;**
  if e_2 **then** P_2 **else skip;**
done

Тогда **while** e_i **do** P_i **done** $\sigma_i \Downarrow \sigma_i'$ *при* $i \in \{1,2\}$ *тогда и только тогда, когда* $P,\sigma_1 \cup \sigma_2 \Downarrow \sigma_1' \cup \sigma_2'$.

Доказательство. ($\Leftarrow$): индукцией по сумме глубин выводов **while** e_i **do** p_i **done** $\sigma_i \Downarrow \sigma_i'$ при $i \in \{1,2\}$.

($\Rightarrow$): индукцией по размеру глубины вывода $P,\sigma_1 \cup \sigma_2 \Downarrow \sigma_1' \cup \sigma_2'$.

Доказательство завершено.

Следующая вспомогательное утверждение показывает корректность классического объединения циклов для циклов без зависимости.

Вспомогательное утверждение 3.12. *Если* **while** e_i **do** p_i **done** $\in Com_{Vi}$ *при* $i \in \{1,2\}$ *и* $V_1 \cap V_2 = \varnothing$, *тогда*

while e_1 **do** P_1 **done**
while e_2 **do** P_2 **done** ~

while $e_1 \vee e_2$ **do**
  if e_1 **then** P_1 **else skip;**
  if e_2 **then** P_2 **else skip;**
done

Доказательство. Пусть P будет левосторонней командой, а P’ – правосторонней.

($\Rightarrow$): Предположим, что $\{\varphi\}P\{\psi\}$, а $\sigma \in \varphi$ и $P',\sigma \Downarrow \sigma'$. Покажем, что $\sigma' \in \psi$. Легко показать, что существуют такие $\sigma_1,\sigma'_1 \in State_{V1}$ и $\sigma_2,\sigma'_2 \in State_{V2}$, при которых $\sigma = \sigma_1 \cup \sigma_2$ и $\sigma' = \sigma_1' \cup \sigma_2'$. По вспомогательному утверждению 3.11 **while** e_1 **do** P_1 **done,** $\sigma_1 \Downarrow \sigma_1'$ и **while** e_2 **do** P_2 **done,** $\sigma_2 \Downarrow \sigma_2'$. Тогда легко показать, что **while** e_1 **do** P_1 **done,** $\sigma \Downarrow \sigma_1' \cup \sigma_2$, а **while** e_2 **do** P_2 **done,** $\sigma_1' \cup \sigma_2 \Downarrow \sigma'$. По E_SEQ имеем $P,\sigma \Downarrow \sigma'$. Согласно предположению, $\sigma' \in \psi$.

($\Leftarrow$): Предположим, что $\{\varphi\}P'\{\psi\}$ и $\sigma \in \psi$, а $P,\sigma \Downarrow \sigma'$. Покажем, что $\sigma' \in \psi$. По E_SEQ имеем **while** e_1 **do** P_1 **done,** $\sigma \Downarrow \sigma''$ и **while** e_2 **do** P_2 **done,** $\sigma'' \Downarrow \sigma'$ для некоторого σ''. Нетрудно показать, что существуют $\sigma_1,\sigma_1',\sigma_1'' \in State_{V1}$ и $\sigma_2,\sigma_2',\sigma_2'' \in State_{V2}$, при которых $\sigma = \sigma_1 \cup \sigma_2$ и $\sigma' = \sigma_1' \cup \sigma_2'$, а $\sigma'' = \sigma_1'' \cup \sigma_2''$. По утверждению 3.10 имеем **while** e_1 **do** P_1 **done,** $\sigma_1 \Downarrow \sigma_1''$ и **while** e_2 **do** P_2 **done,** $\sigma_2'' \Downarrow \sigma_2'$, $\sigma_2 = \sigma_2''$ и $\sigma_1'' = \sigma_1'$. По вспомогательному утверждению 3.11 $P',\sigma_1 \cup \sigma_2'' \Downarrow \sigma_1'' \cup \sigma_2'$. По предположению $\sigma_1'' \cup \sigma_2' \in \psi$. Тогда $\sigma' = \sigma_1' \cup \sigma_2' = \sigma_1'' \cup \sigma_2'$ завершает доказательство.

4. Выводы

Представлена технология объединения циклов для повышения эффективности статической верификации программ. Технология основана на подходе «предположения и допущения», который позволяет объединить два удаленных цикла, имеющих

зависимость данных. Формализован алгоритм и планируется доказательство его корректности для простого императивного языка. Будет разработан метод предобработки для объединения циклов, основанный на обращении выполнения циклов.

Перспективное приложение – это *верификация невмешательства с использованием самокомпозиции* [9]. Проблема верификации невмешательства программы P в [5, 7, 8] сводится к некоторой проблеме верификации безопасности самосоставной программы P; P′ где P′ – программа, полученная путем «штрихования» всех переменных в P. Если P содержит цикл, то самосоставная программа содержит циклы со сходными шаблонами циклов, для которых предложенная технология эффективна.

ЛИТЕРАТУРА/ REFERENCES

1. Bacon D.F., Graham S.L., Sharp O.J. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 1994;26(4):345-420.
2. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. *CAV 2011 (LNCS). Springer*. 2011;6806:184-190.
3. Cohen E. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*. *ACM*. 1977;11:133-139.
4. Ferrante J., Ottenstein K.J., Warren J.D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 1987;9(3):319-349.
5. Goguen J.A., Meseguer J. Security Policies and Security Models. *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA.* 1982:11-20.
6. Gurfinkel A., Kahsai T., Komuravelli A., Navas J.A. The SeaHorn verification framework. *CAV 2015 (LNCS), Vol. 9206. Springer*. 2015:343-361.
7. Li P., Zdancewic S. Downgrading policies and relaxed noninterference. *Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA.* 2005:158-170.
8. McLean J. Security Models and Information Flow. *Proc. of the IEEE Symposium on Security and Privacy, Oakland, California, USA.* 1990:180-189.
9. Terauchi T., Aiken A. Secure Information Flow as a Safety Problem. *Static Analysis, 12th Int. Symposium, SAS 2005, London, UK*. 2005:352-367.
10. Winskel G. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press. 1993.
11. Barnett M., Chang B.-Y.E., DeLine R., Jacobs B., Leino K.R.M. Boogie: *A Modular Reusable Verifier for Object-Oriented Programs.* FMCO 2005 (LNCS). 2005;4111:364-387.
12. Reynolds J.C. *Separation Logic: A Logic for Shared Mutable Data Structures*. Proc. of IEEE Symposium on Logic In Computer Science (LICS'02). 2002.

ИНФОРМАЦИЯ ОБ АВТОРЕ / INFORMATIONS ABOUT AUTHORS

Лысов Дмитрий Вячеславович, аспирант, Воронежский государственный технический университет, Воронеж, Российская Федерация.
csit@bk.ru

Dmitry V. Lysov, graduate student, Voronezh State Technical University, Voronezh, Russian Federation.