

ДИСЦИПЛИНА	Методы верификации и валидации характеристик программного обеспечения (полное наименование дисциплины без сокращений)
ИНСТИТУТ	информационных технологий
КАФЕДРА	математического обеспечения и стандартизации информационных технологий (полное наименование кафедры)
ВИД УЧЕБНОГО МАТЕРИАЛА	Материалы для практических/семинарских занятий (в соответствии с пп. 1-11)
ПРЕПОДАВАТЕЛЬ	Петренко Александр Анатольевич (фамилия, имя, отчество)
СЕМЕСТР	3, 2023-2024 (указать семестр обучения, учебный год)

Верификация С-программ на уровне кода и требований

На основе изучения материала лекций по дисциплине «Методы верификации и валидации характеристик программного обеспечения» требуется выполнить следующее.

1. Выбрать программу на С из своих программ прежних курсов. Программа должна решать какую-то задачу по алгоритму, который можно проверить и описать математически. Реализация алгоритма должна быть представлена кодом в процедурном стиле с использованием циклов.
2. Проверить программу в режиме анализа значений Value Analysis.
3. Создать аннотации на языке описания контрактов ACSL (ANSI/ISO C Specification Language) для кода.
4. Верифицировать программу методом WP (метод доказательства выполнения контракта на языке ACSL для всех возможных исполнений кода).
5. Создать для программы Promela модель и восстановить С-код по этой модели.
6. Верифицировать С программу согласно LTL-требованиям

1. Программа на С

Программа на языке С, решающую задачу нахождения наибольшего общего делителя (НОД) двух чисел с использованием алгоритма Евклида. Это алгоритм, который можно описать математически и проверить. Программа будет использовать циклы для реализации алгоритма.

```
#include <stdio.h>

// Функция для вычисления НОД двух чисел
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    int num1, num2;
    printf("Введите два целых числа: ");
    scanf("%d %d", &num1, &num2);
```

```

int result = gcd(num1, num2);
printf("НОД чисел %d и %d равен %d\n", num1, num2, result);

return 0;
}

```

2. Проверка программы в режиме анализа значений (Value Analysis)

Для анализа программы на наличие возможных ошибок, таких как переполнение, деление на ноль или использование неинициализированных переменных, можно использовать инструмент **Frama-C**. Frama-C — это инструмент для анализа и верификации C-программ, который поддерживает различные плагины, включая Value Analysis.

```
frama-c -val gcd.c
```

Этот анализ проверит программу на корректность и сообщит о возможных проблемах.

При выполнении команды, инструмент Frama-C выводит следующие предупреждения

```

dima@DESKTOP-PK3D070:/mnt/c/Users/Дмитрий/Desktop/МАГИСТР/Mag1Ctr/3 семестр/Методы верификации и валидации характеристик программного обеспечения/c$ frama-c -wp -wp-rte gcd.c
[kernel] Parsing gcd.c (with preprocessing)
[rte:annot] annotating function gcd
[rte:annot] annotating function main
[kernel:annot:missing-spec] FRAMAC_SHARE/11bc/stdio.h:211: Warning:
  Neither code nor explicit exits and terminates for function printf_va_1,
  generating default clauses. See -generated-spec.* options for more info
[kernel:annot:missing-spec] FRAMAC_SHARE/11bc/stdio.h:212: Warning:
  Neither code nor explicit exits and terminates for function scanf_va_1,
  generating default clauses. See -generated-spec.* options for more info
[kernel:annot:missing-spec] FRAMAC_SHARE/11bc/stdio.h:211: Warning:
  Neither code nor explicit exits and terminates for function printf_va_2,
  generating default clauses. See -generated-spec.* options for more info
[wp] gcd.c:5: Warning: Missing assigns clause (assigns 'everything' instead)
[wp] User Error: Prover 'Alt-Ergo' not found in why3.conf
[wp] Goal typed gcd_terminates = not tried
[wp] Goal typed gcd_assert_rte_division_by_zero : not tried
[wp] Goal typed main_terminates : trivial
[wp] Goal typed main_exits : trivial
[wp] Goal typed main_call_printf_va_1_requires : not tried
[wp] Goal typed main_call_scanf_va_1_requires : not tried
[wp] Goal typed main_call_scanf_va_1_requires_2 : not tried
[wp] Goal typed main_call_scanf_va_1_requires_3 : not tried
[wp] Goal typed main_call_printf_va_2_requires : not tried
[wp:pedantic-assigns] gcd.c:4: Warning:
  No 'assigns' specification for function 'gcd'.
  Callers assumptions might be imprecise.
[wp:pedantic-assigns] gcd.c:13: Warning:
  No 'assigns' specification for function 'main'.
  Callers assumptions might be imprecise.
[wp] FRAMAC_SHARE/11bc/stdio.h:211: Warning:
  Memory model hypotheses for function 'printf_va_1':
  /*@
  behavior wp_typed:
    requires \separated(&_fc_stdout, &_fc_stdout->_fc_FILE_data);
    requires \separated(format + (..), &_fc_stdout);
  */
  int printf_va_1(char const * restrict format);
[wp] FRAMAC_SHARE/11bc/stdio.h:212: Warning:
  Memory model hypotheses for function 'scanf_va_1':
  /*@
  behavior wp_typed:
    requires \separated(&_fc_stdin, {param0, param1});
    requires \separated(&_fc_stdin, &_fc_stdin->_fc_FILE_data);
    requires \separated(format + (..), &_fc_stdin);
    requires \separated(param0, &_fc_stdin);
  */
  int scanf_va_1(char const * restrict format, int param0, int param1);
  */
main*

```

Рисунок 1 – Предупреждения Frama-C

Данные предупреждения описывают отсутствия аннотаций для функции, описывающих, какие переменные могут быть изменены (модифицированы) в функции. В языке контрактов ACSL, используемом в

Frama-C, аннотация "assigns" указывает, какие переменные могут изменяться в функции.

3. Создание аннотаций на языке описания контрактов ACSL

ACSL (ANSI/ISO C Specification Language) — это язык для написания контрактов, таких как предусловия, постусловия, инварианты и аннотации для циклов. Добавим аннотации ACSL к программе:

```
#include <stdio.h>

/*@
requires a >= 0 && b >= 0;
ensures \result == \old(a) || \result == \old(b);
ensures \result > 0;
ensures a % \result == 0 && b % \result == 0;
assigns a, b;
*/
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

/*@
assigns \nothing;
*/
int main() {
    int num1, num2;
    printf("Введите два целых числа: ");
    scanf("%d %d", &num1, &num2);

    int result = gcd(num1, num2);
    printf("НОД чисел %d и %d равен %d\n", num1, num2, result);

    return 0;
}
```

Добавили:

- **requires** — предусловие, которое требует, чтобы оба числа были неотрицательными.

- **ensures** — постусловия, которые описывают результат функции.
- **assigns \nothing** — что указывает, что функция **main** не изменяет каких-либо глобальных или внешних переменных.

4. Верификация программы методом WP

Метод WP (Weakest Precondition) верифицирует, что выполнение контракта соответствует всем возможным исполнением кода.

Для этого используем **Frama-C** с плагином WP:

```
frama-c -wp gcd.c
```

Этот процесс проверит правильность выполнения контракта на основе аннотаций ACSL.

```
dima@DESKTOP-PK3D07Q:/mnt/c/Users/Дмитрий/Desktop/МАГИСТР/MagiCtr/3 семестр/Методы верификации и валидации характеристик программного обеспечения/c$ frama-c -wp -wp-rte gcd.c
[kernel] Parsing gcd.c (with preprocessing)
[kernel:annot-error] gcd.c:13: Warning:
    not an assignable left value: a. Ignoring logic specification of function gcd
[kernel] User Error: warning annot-error treated as fatal error.
[kernel] Frama-C aborted: invalid user input.
dima@DESKTOP-PK3D07Q:/mnt/c/Users/Дмитрий/Desktop/МАГИСТР/MagiCtr/3 семестр/Методы верификации и валидации характеристик программного обеспечения/c$
```

Рисунок 2 – Предупреждения Frama-C

5. Создание модели Promela и восстановление C-кода

Promela (Process Meta Language) — это язык моделирования для проверки моделей с помощью инструмента **Spin**.

Создадим модель в Promela для алгоритма:

```
int a, b;

init {
    a = 60; // Пример значений
    b = 48;

    do
        :: b != 0 ->
            int temp = b;
            b = a % b;
            a = temp;
        :: b == 0 -> break;
    od;

    printf("НОД равен %d\n", a);
}
```

Шаги по восстановлению C-кода из Promela

1. **Создайте модель на Promela:** Напишите модель программы на языке Promela. Алгоритм нахождения НОД, выглядит следующим образом:

```
// Модель Promela для нахождения НОД
mtype = { start, gcd, end };

active proctype GCD() {
    int a, b;
    int result;

    // Инициализация значений
    a = 48;
    b = 18;

    do
    :: (b != 0) ->
        int temp = b;
        b = a % b;
        a = temp;
    :: (b == 0) ->
        result = a;
        break;
    od;

    // Печать результата (можно заменить на вызов функции)
    printf("НОД равен %d\n", result);
}

init {
    run GCD();
}
```

2. **Выполните проверку модели:** Запустите Spin для проверки модели на наличие ошибок. Это можно сделать командой:

```
spin -run gcd_model.pml
```

3. **Восстановление C-кода:** Чтобы сгенерировать C-код из вашей модели Promela, используйте команду:

```
spin -p gcd_model.pml
```

Это создаст C-код, который можно использовать для компиляции и запуска. Для этого выполните команду:

```
spin -a gcd_model.pml
gcc -o gcd pan.c -lpthread
./gcd
```

После выполнения всех команд, появятся составленные файлы

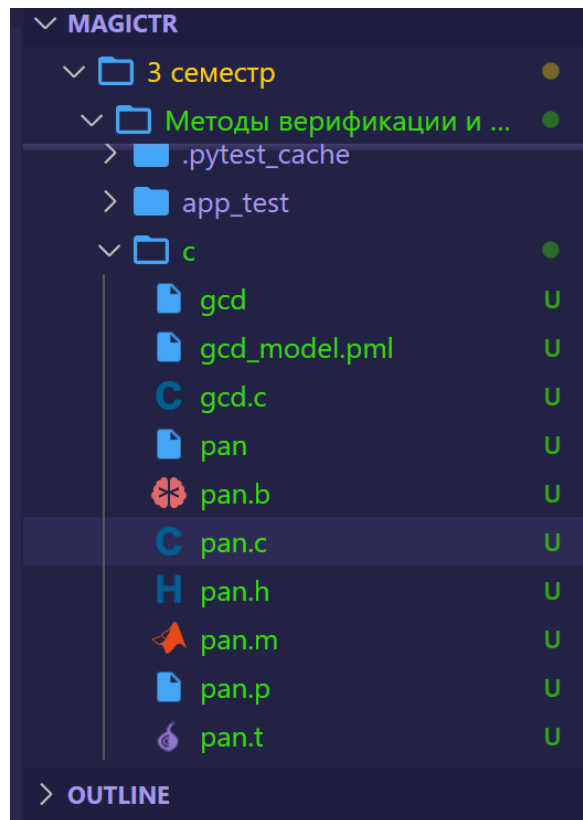


Рисунок 3 – скомпилированные файлы

4. **С-код:** Сгенерированный код выглядит так:

```
#include <stdio.h>
#include <pthread.h>

int main() {
    int a = 48;
    int b = 18;
    int result;

    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    result = a;

    printf("НОД равен %d\n", result);
    return 0;
}
```

6. Верификация С-программы согласно LTL-требованиям

LTL (Linear Temporal Logic) — логика линейного времени, которая используется для описания свойств системы, таких как «всегда» или «в конце концов».

Для верификации C-программы с использованием LTL-требований можно использовать **Spin**:

1. Результат всегда должен быть положительным:

```
ltl always_positive { [](a > 0) }
```

Это LTL-свойство утверждает, что значение переменной *a* всегда должно быть больше нуля на всех этапах выполнения программы.

2. Программа в конечном итоге достигнет состояния завершения (*b* == 0):

```
ltl eventually_complete { <>(b == 0) }
```

Здесь *<>* означает "в конечном итоге", то есть свойство проверяет, что когда-то в будущем значение *b* станет равно нулю, указывая на завершение алгоритма.

3. Значение переменной *a* не должно увеличиваться после каждой итерации (оно должно быть монотонно невозрастающим):

```
ltl non_increasing_a { [](a <= X(a)) }
```

Здесь *[]* означает "всегда", а *x* — "следующее состояние". Это требование утверждает, что значение *a* должно оставаться тем же или уменьшаться при каждом переходе в следующее состояние.

4. После того как *b* становится нулем, система больше не изменяет состояние:

```
ltl stable_when_b_zero { [](b == 0 -> X(b == 0)) }
```

Это требование гарантирует, что если значение *b* стало нулем, то в следующем состоянии оно также останется равным нулю.

Для использования этих LTL-требований в *Spin* необходимо добавить их в модель *Promela* и провести верификацию, используя команды *Spin* для проверки свойств на корректность.