

Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

Институт искусственного интеллекта   
Базовая кафедра №252 – информационной безопасности

**практическая работа по предмету**

**«Методы верификации»**

|  |  |  |
| --- | --- | --- |
| **Студент группы ККСО-01-20** |  | *Семин В.В.* |
| **Руководитель** |  | *Старший преподаватель*  *Коробов Даниил*  *Владимирович* |

ОГЛАВЛЕНИЕ

[1 СТАТИЧЕСКИЙ АНАЛИЗ КОДА С ПОМОЩЬЮ PVS-STUDIO 3](#_Toc188218980)

[1.1 ФАЙЛ CWE562\_RETURN\_OF\_STACK\_VARIABLE\_ADDRESS\_\_RETURN\_BUF\_01.C 3](#_Toc188218981)

[1.2 ФАЙЛ CWE562\_RETURN\_OF\_STACK\_VARIABLE\_ADDRESS\_\_RETURN\_LOCAL\_CLASS\_MEMBER\_01.CPP 4](#_Toc188218982)

[2 ВЕРИФИКАЦИЯ ФУНКЦИИ В FRAMA-C 7](#_Toc188218983)

[2.1 ВЕРИФИКАЦИЯ ФУНКЦИИ SWAP 7](#_Toc188218984)

[2.2 ВЕРИФИКАЦИЯ ФУНКЦИИ INDEX\_OF\_MIN\_ELEMENT 8](#_Toc188218985)

[2.2.1 Предусловия и постусловия функции index\_of\_min\_element 8](#_Toc188218986)

[2.2.2 Верификация цикла функции index\_of\_min\_element 10](#_Toc188218987)

[2.3 ВЕРИФИКАЦИЯ ФУНКЦИИ SELECTION\_SORT 12](#_Toc188218988)

[2.3.1 Верификация постусловия неубывания элементов массива 12](#_Toc188218989)

[2.3.2 Индуктивное определение перестановки элементов массива 14](#_Toc188218990)

[2.3.3 Верификация постусловия сохранения элементов массива 15](#_Toc188218991)

[2.3.4 Вспомогательные конструкции внутри блока цикла 17](#_Toc188218992)

[2.4 РЕЗУЛЬТАТ 19](#_Toc188218993)

[СПИСОК ЛИТЕРАТУРЫ 20](#_Toc188218994)

[ПРИЛОЖЕНИЕ А 21](#_Toc188218995)

[ПРИЛОЖЕНИЕ Б 26](#_Toc188218996)

# 1 СТАТИЧЕСКИЙ АНАЛИЗ КОДА С ПОМОЩЬЮ PVS-STUDIO

Juliet Test Suite — это набор тестов, разработанный для анализа и оценки программного обеспечения на наличие уязвимостей безопасности. Он широко используется для оценки инструментов анализа кода [1].

В данной работе будет выбраны два тестовых файла из данного набора, реализующих уязвимость CWE562. Она заключается в следующем.

Поскольку локальные переменные выделяются в стеке, когда программа возвращает указатель на локальную переменную, она возвращает адрес из стека. Последующий вызов функции, скорее всего, заново использует этот же адрес стека, перезаписывая значение указателя, который больше не будет соответствовать той же самой переменной, так как стековый фрейм функции становится недействительным после её завершения. В лучшем случае это приведёт к неожиданному изменению значения указателя. В большинстве случаев это приводит к аварийному завершению программы при следующей попытке разыменования указателя [2].

Для анализа данных тестовых случаев будет использован инструмент статического анализа PVS-Studio. Наиболее удобным вариантом изучения данного инструмента в лабораторных условиях будет использование его версии, интегрированной в инструмент для исследования компиляторов Compiler Explorer.

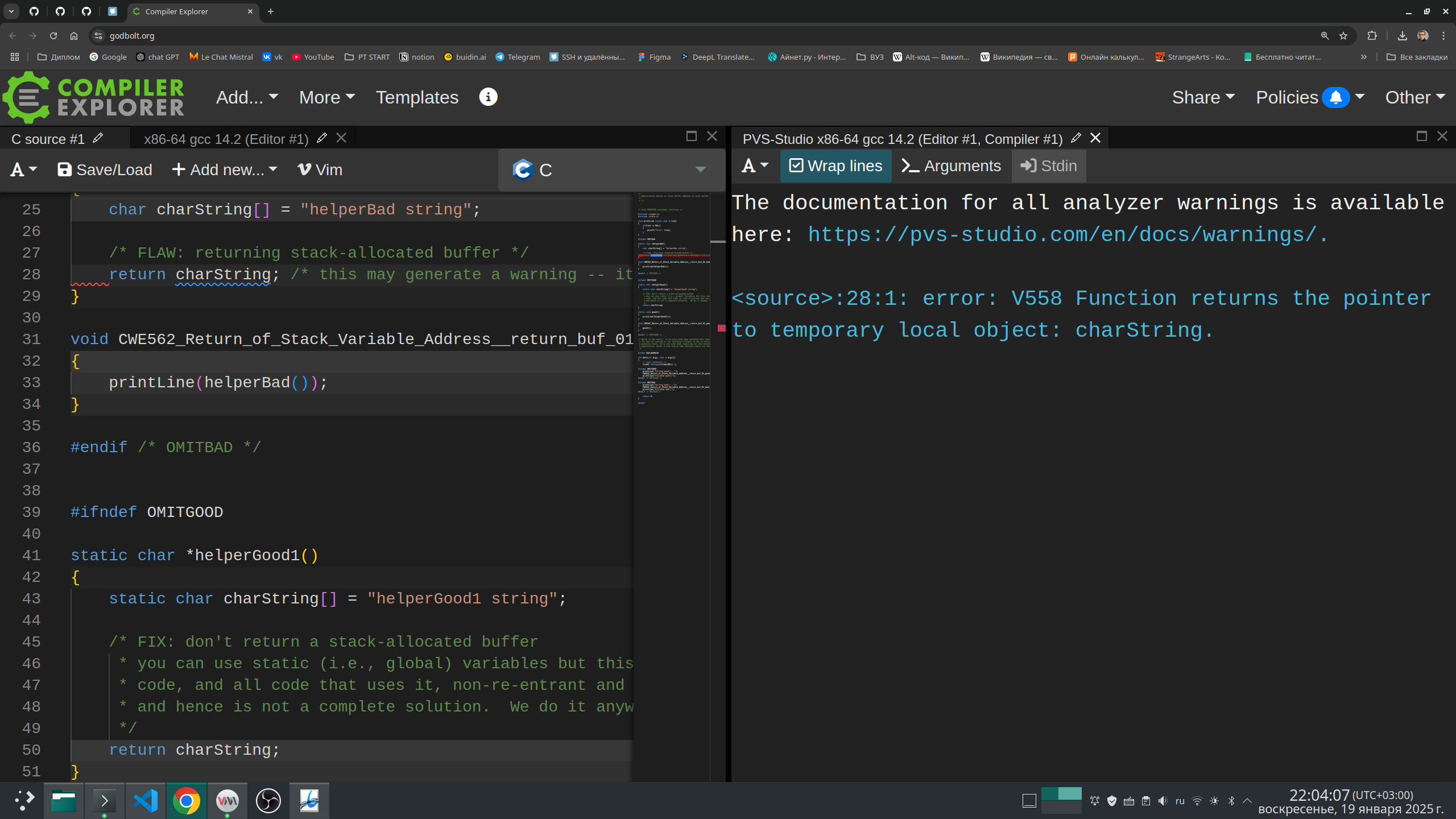
## 1.1 ФАЙЛ CWE562\_RETURN\_OF\_STACK\_VARIABLE\_ADDRESS\_\_RETURN\_BUF\_01.C

Рассмотрим файл *CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01.c* (Листинг А.1 в Приложении А).

Данный код имеет две тестовые функции. В первой возвращаемое значение функции определено с помощью ключевого слова *static*. Это означает, что память для массива выделяется один раз при первом вызове функции, и строка сохраняет свое значение между вызовами функции. Следовательно, возвращение адреса данного массива не вызовет ошибок.

Во второй данное ключевое слово пропущено. Следовательно, адрес массива может быть перезаписан, что приведёт к ошибке.

Проверим данный файл в PVS-Studio (Рисунок 1.1).



*Рисунок 1.1 – Анализ файла CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01.c*

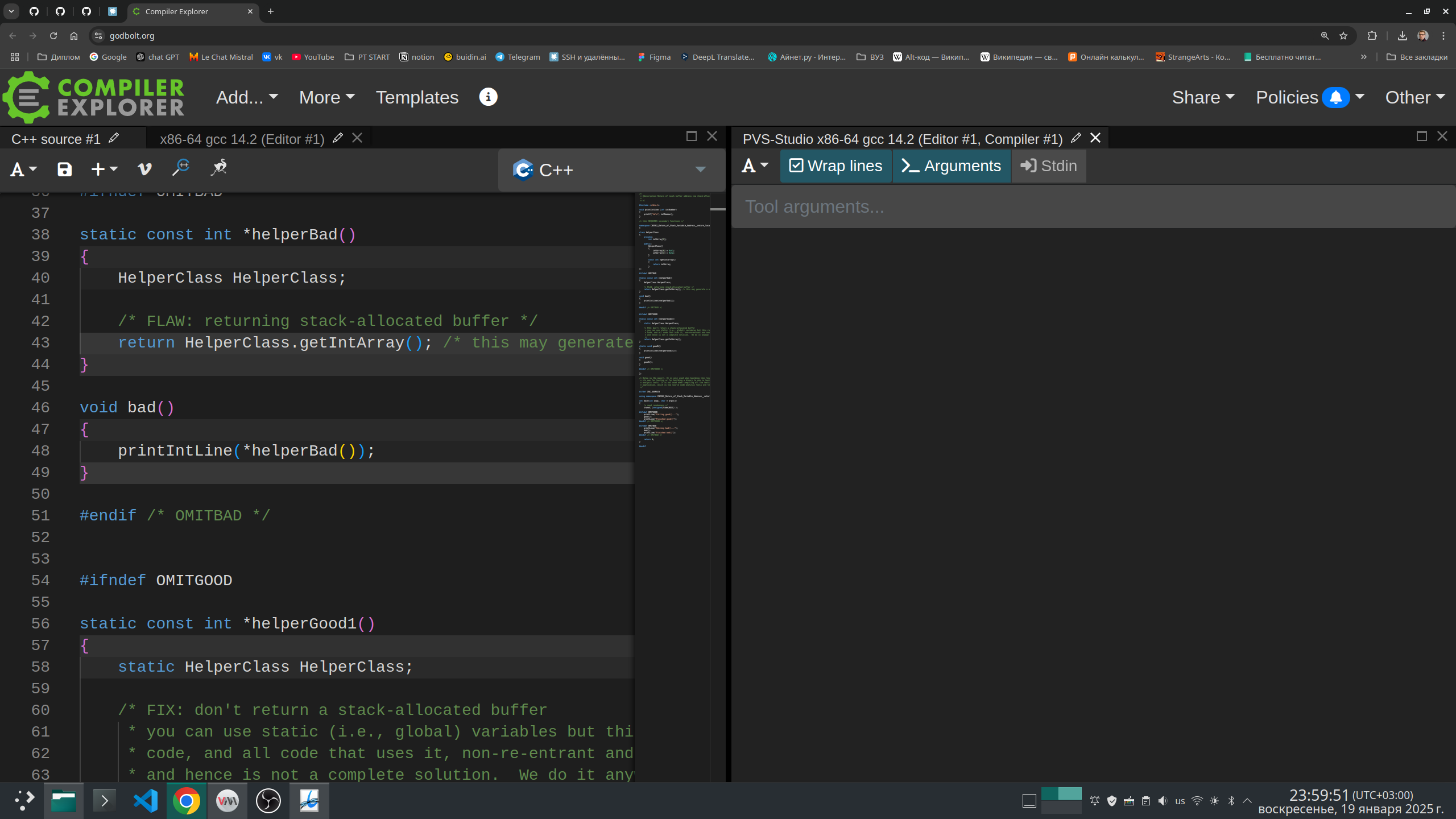
Инструмент обнаружил ошибку в неисправной функции и при этом не дал ложноположительного ответа для исправной. Следовательно, он корректно проанализировал данный случай.

## 1.2 ФАЙЛ CWE562\_RETURN\_OF\_STACK\_VARIABLE\_ADDRESS\_\_RETURN\_LOCAL\_CLASS\_MEMBER\_01.CPP

Рассмотрим файл *CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_local\_class\_member\_01.cpp* (Листинг А.2 в Приложении А).

Данный тестовый пример во многом аналогичен первому. Есть две функции, в одной из которых переменная объявлена с ключевым словом *static*, в другой это слово пропущено. Однако этот случай является более сложным по сравнению с первым из-за, того, что возвращаемым значением функции является не сама локальная переменная, а одно из её полей.

Проверим данный файл в PVS-Studio (Рисунок 1.2).



*Рисунок 1.2 – Анализ файла CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_local\_class\_member\_01.cpp*

Инструмент не смог обнаружить ошибку в данном файле. Это может быть связанно с двумя причинами.

Первая причина – ограничения самого инструмента. Представленная в Compiler Explorer версия PVS-Studio работает в режиме общей диагностики [3]. Возможно, что в данном режиме инструмент ориентирован на выявление типичных ошибок и не всегда глубоко анализирует контексты использования указателей.

Вторая причина – неуточнённое поведение. Программа имеет неуточнённое поведение, когда стандарт предоставляет две или более возможности, но не даёт рекомендаций, какие из них должны быть выбраны автором компилятора [4]. В данном случае ошибка анализа может произойти из-за того, что разные компиляторы и даже один компилятор с разными флагами могут отлично друг от друга обрабатывать область стека, которая была выделена под вызов функции, после её завершения.

# 2 ВЕРИФИКАЦИЯ ФУНКЦИИ В FRAMA-C

Рассмотрим код функции сортировки выбором (Листинг 2.1).

*Листинг 2.1 – Сортировка выбором*



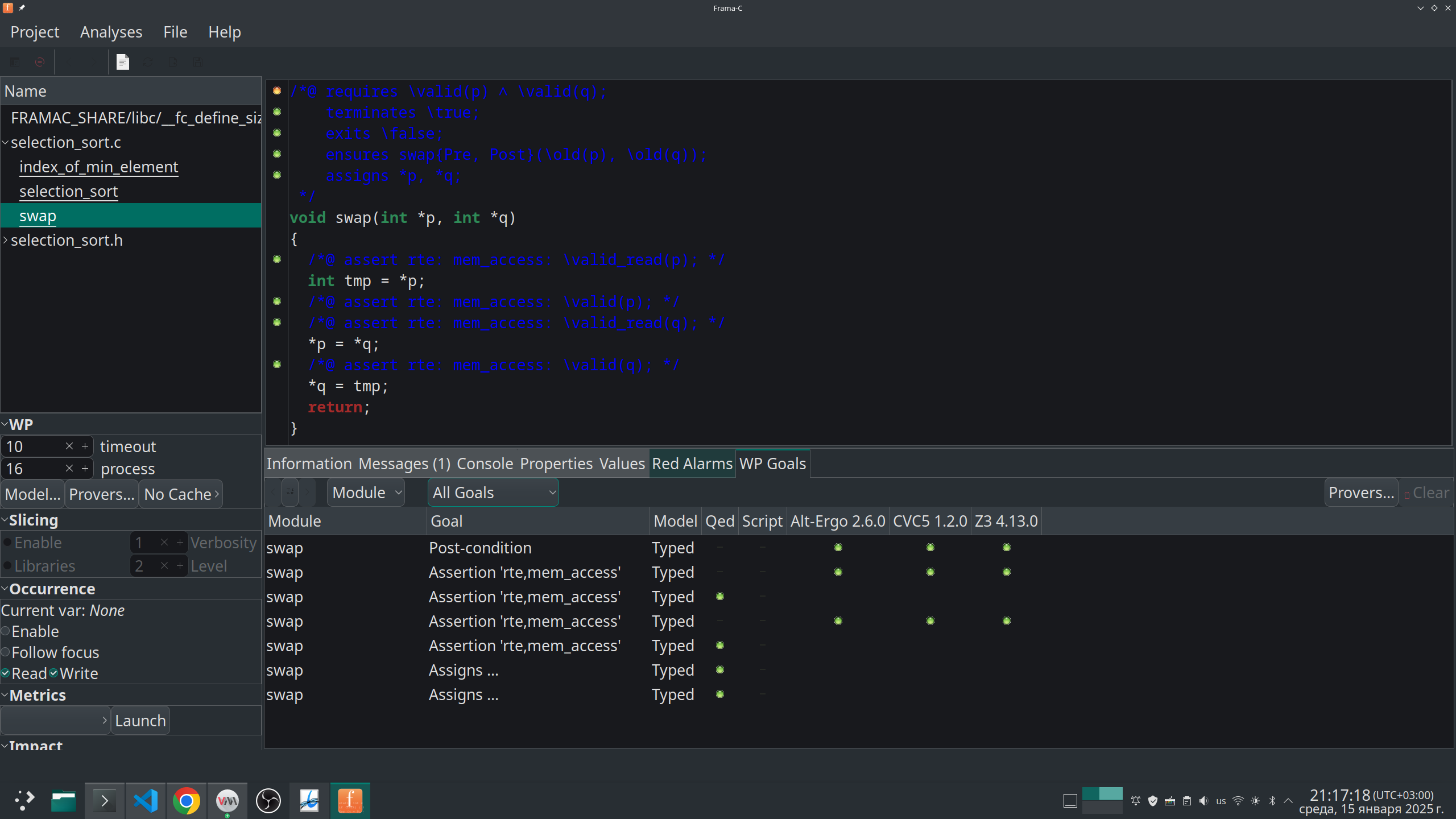
## 2.1 ВЕРИФИКАЦИЯ ФУНКЦИИ SWAP

Начнём с верификации функции обмена значений переменных. Для корректной работы функции необходимо, чтобы ячейки памяти, на которые ссылаются указатели *p* и *q*, были доступны для чтения и записи. Постусловием работы функции *swap* является равенство значения, на которое ссылается *q*, значению, на которое *p* ссылалось изначально, и равенство значения, на которое ссылается *p*, значению, на которое *q* ссылалось изначально. Также стоит учесть побочный эффект работы функции. Единственными внешними для неё ячейками памяти, которая она может изменить, являются те, на которые указывают *p* и *q*. Получаем следующий контракт (Листинг 2.2).

*Листинг 2.2 – Контракт функции swap*



Запустим Frama-C и проверим, верифицируется ли данный контракт (Рисунок 2.1).



*Рисунок 2.1 – Верификация функции swap*

Постусловия функции успешно верифицированы. Предусловия на данный момент не верифицируются, так как ещё не был написан контракт функции *selection\_sort*, которая выполняет вызов *swap*.

## 2.2 ВЕРИФИКАЦИЯ ФУНКЦИИ INDEX\_OF\_MIN\_ELEMENT

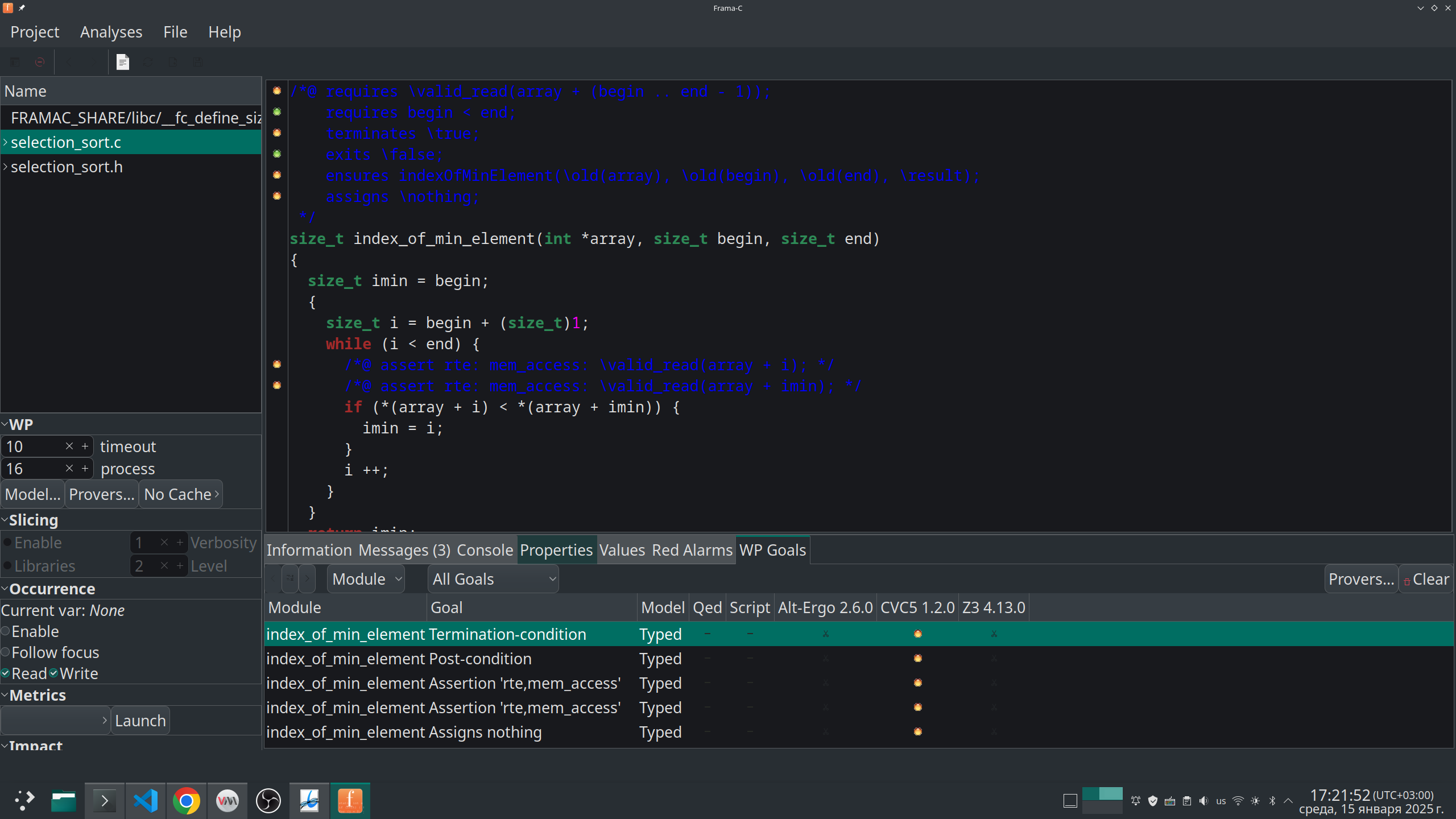
### 2.2.1 Предусловия и постусловия функции index\_of\_min\_element

Далее рассмотрим функцию, которая выполняет поиск минимального элемента в массиве. Для её корректного выполнения необходимо, чтобы массив был доступен для чтения, он был не пустым (то есть его конечный адрес был больше начального). Постусловием функции является то, что результат её работы – индекс элемента, значение которого не больше значений остальных элементов функции. При этом этот индекс должен находиться внутри массива. Также стоит учесть, что функция не должна изменять никакие внешние для неё ячейки памяти. Получаем следующий контракт (Листинг 2.3).

*Листинг 2.3 – Контракт функции index\_of\_min\_element*



Запустим Frama-C и проверим, верифицируется ли данный контракт (Рисунок 2.2).

*Рисунок 2.2 – Неуспешная верификация функции index\_of\_min\_element*

Постусловия функции на данный момент не верифицируются. Связано это с тем, что внутри данной функции находится цикл. Верификация цикла является сложной задачей, с которой модуль дедуктивной верификации wp самостоятельно справится не может.

### 2.2.2 Верификация цикла функции index\_of\_min\_element

Для успешной верификации цикла необходимо найти свойство, которое будет верно перед началом цикла и после любой его итерации. Такое условие называется инвариант цикла.

Кроме этого, wp не может заранее рассчитать количество итераций в цикле и, следовательно, верифицировать его конечность. Для этого следует задать значение, которое будет убывать после каждой итерации. Такое значение называется вариант цикла. Вариант задаёт верхнюю границу количества итераций цикла. Вариант обязательно должен быть неотрицательным в начале каждой итерации, но может стать отрицательным в последней итерации.

Также стоит учесть, что для цикла, также как и для самой функции, следует задать перечень внешних для него ячеек памяти, которые он может изменять. Счётчик цикла *for* также является внешней для него переменной, так как *for* при верификации преобразуется в *while* [5].

Учитывая всё вышесказанное, напишем спецификацию для цикла внутри функции *index\_of\_min\_element* (Листинг 2.4).

*Листинг 2.4 – Спецификация цикла функции index\_of\_min\_element*



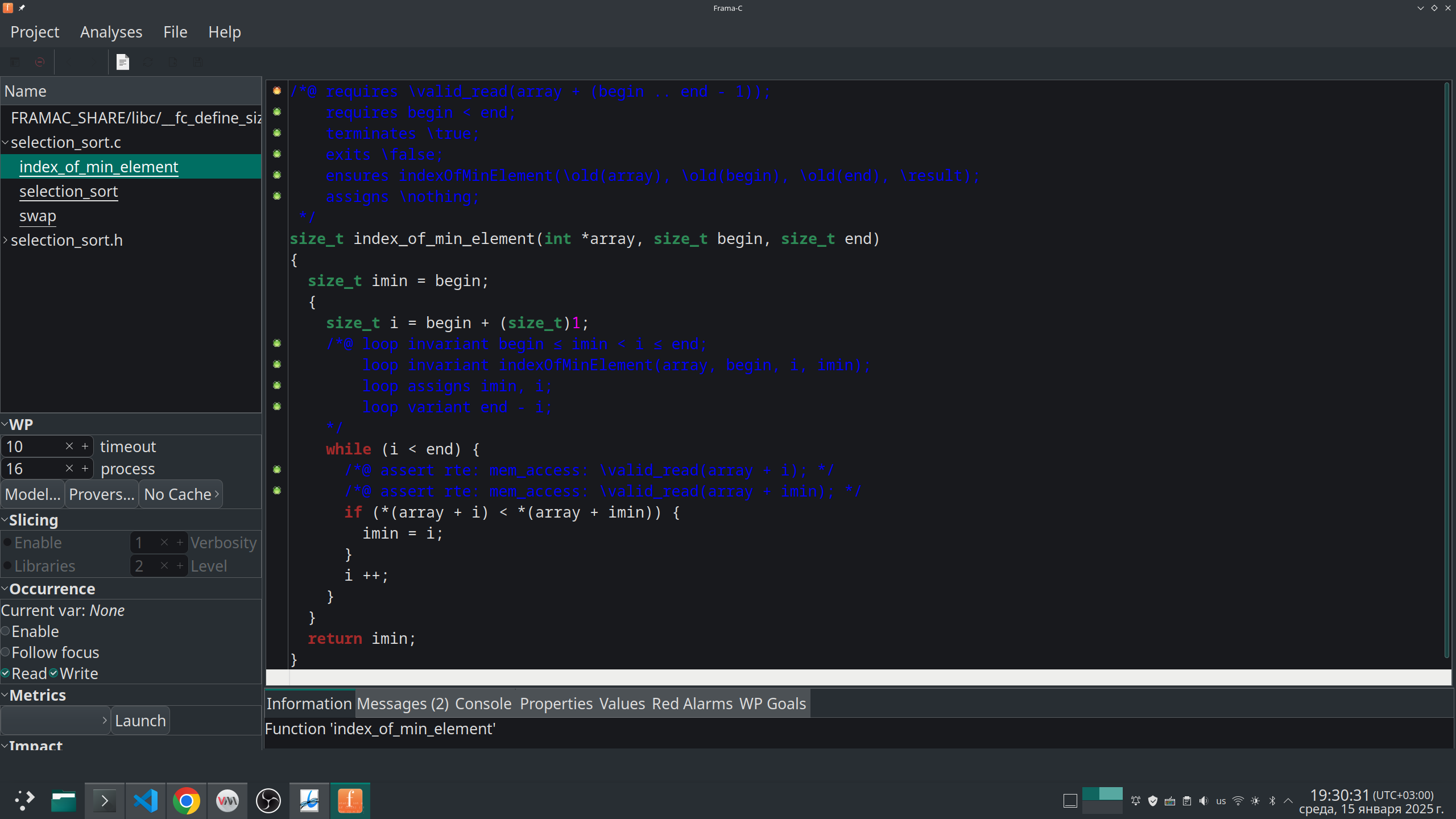
В данном случае первая часть инварианта задаёт количество раз, которое цикл изменяет переменные, вторая часть связывает инвариант с постусловием контракта функции, что необходимо для успешной её верификации.

Обратим внимание, что вариант в данном случае не задаёт точное количество оставшихся итераций цикла. Это не является необходимостью. Главное, чтобы выполнялись вышеописанные ограничения на вариант.

В перечне изменяемых циклом переменных заданы индекс минимального элемента, так как данная переменная инициализирована до цикла, и счётчик цикла, так как уже было упомянуто, что он является внешней переменной для цикла.

Для того, чтобы не ошибиться при верификации цикла, следует помнить о том, что для любого значения в *loop assigns* должно существовать условие в инварианте цикла.

Теперь постусловия функции успешно верифицируются (Рисунок 2.3).



*Рисунок 2.3 – Успешная верификация функции index\_of\_min\_element*

## 2.3 ВЕРИФИКАЦИЯ ФУНКЦИИ SELECTION\_SORT

### 2.3.1 Верификация постусловия неубывания элементов массива

Для корректной работы функции сортировки необходимо, чтобы массив был доступен для чтения. Также, чтобы успешно верифицировать вызов функцию поиска индекса минимального элемента, в предусловиях функции сортировки стоит учесть её предусловие о непустом массиве. Постусловием функции будет не убывание элементов массива. Функция должна изменять только память массива. С учётом вышесказанного получим следующий контракт функции сортировки (Листинг 2.5).

*Листинг 2.5 – Контракт функции selection\_sort с постусловием неубывания элементов массива*



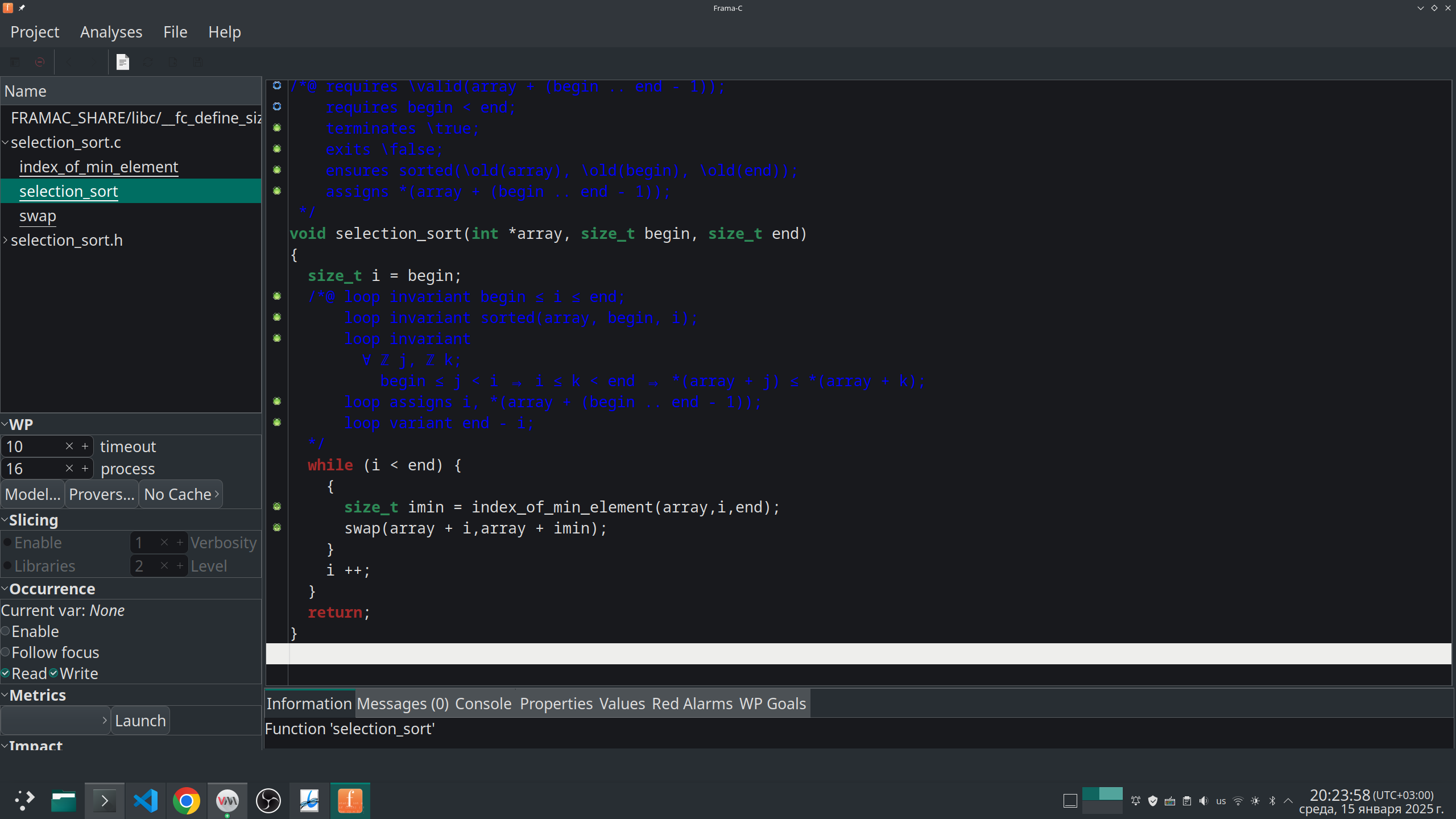
Не стоит забывать и о спецификации цикла (Листинг 2.6).

*Листинг 2.6 – Спецификация цикла функции selection\_sort с инвариантом не убывания элементов массива*



В данном случае необходимо три условия в инварианте. Первое говорит о количестве внесённых в массив изменений. Второе связывает инварианту с постусловием. Третье необходимо, так как первые два не описывают состояние элементов массива между индексами *i* и *end* (напоминаем, что в корректной спецификации для любой переменной в перечне *loop assigns* должно существовать условие в инварианте функции).

Верификация проходит успешно (Рисунок 2.4).



*Рисунок 2.4 – Верификация постусловия не убывания элементов массива функции selection\_sort*

### 2.3.2 Индуктивное определение перестановки элементов массива

На данный момент контракт функции *selection\_sort* является неполным. Он не учитывает то, что сортировка должна сохранить элементы массива, то есть отсортированный массив должен быть перестановкой изначального массива.

Для этого необходимо ввести предикат перестановки. Наиболее удобным будет его индуктивное определение. Индукция в ACSL аналогична индукции в математике. Сначала описываются базовые случаи, а потом переход от случая с меньшим значением к случаю с большим значением. В данном случае индукция будет по количеству обменов значениями между элементами массива (по сути, количеству вызовов функции *swap*).

Для описания базового случая введём предикат *swapInArray*, показывающий ситуацию, когда два элемента массива обменялись значениями, а остальные остались неизменными (Листинг 2.7).

*Листинг 2.7 – Предикат swapInArray*



Теперь введём индуктивное определение предиката *permutation*. Оно будет состоять из трёх случаев: массив является перестановкой на самом себе (свойство рефлексивности отношения быть перестановкой), *swapInArray* является перестановкой, перестановка перестановки тоже является перестановкой изначального массива (свойство транзитивности отношения быть перестановкой). Получим следующую спецификацию (Листинг 2.8).

*Листинг 2.8 – Предикат permutation*



### 2.3.3 Верификация постусловия сохранения элементов массива

Дополним контракт *selection\_sort* новым предикатом (Листинг 2.9).

*Листинг 2.9 – Контракт функции selection\_sort с постусловием сохранения элементов массива*

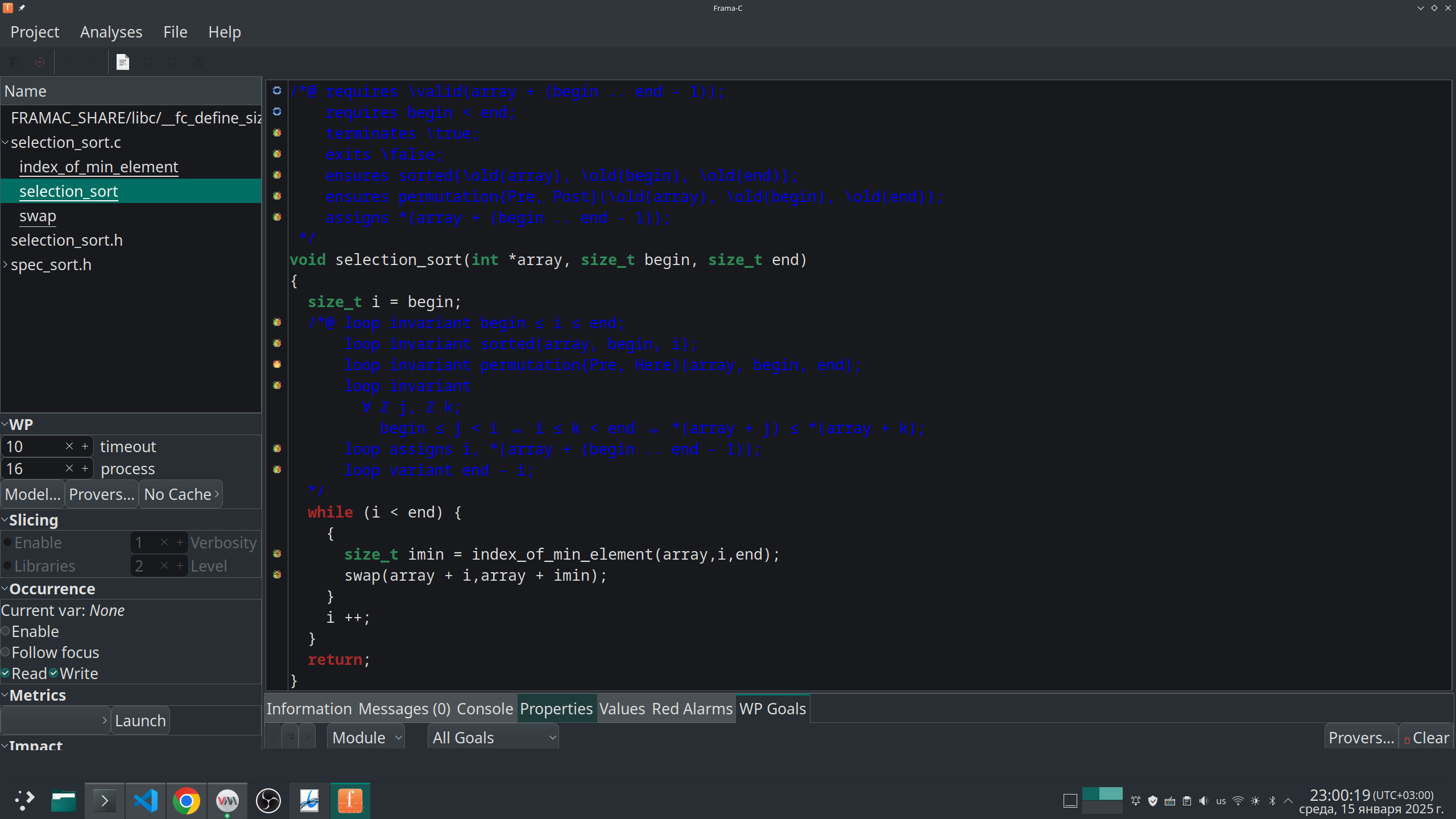
**

Также добавим новое условие в инвариант цикла, которая свяжет её с новым постусловием (Листинг 2.10).

*Листинг 2.10 – Спецификация цикла функции selection\_sort с инвариантом сохранения элементов массива*

**

Проверим, верифицирует ли Frama-C новое постусловие (Рисунок 2.5).



*Рисунок 2.5 – Неуспешная верификация постусловия сохранения элементов массива функции selection\_sort*

### 2.3.4 Вспомогательные конструкции внутри блока цикла

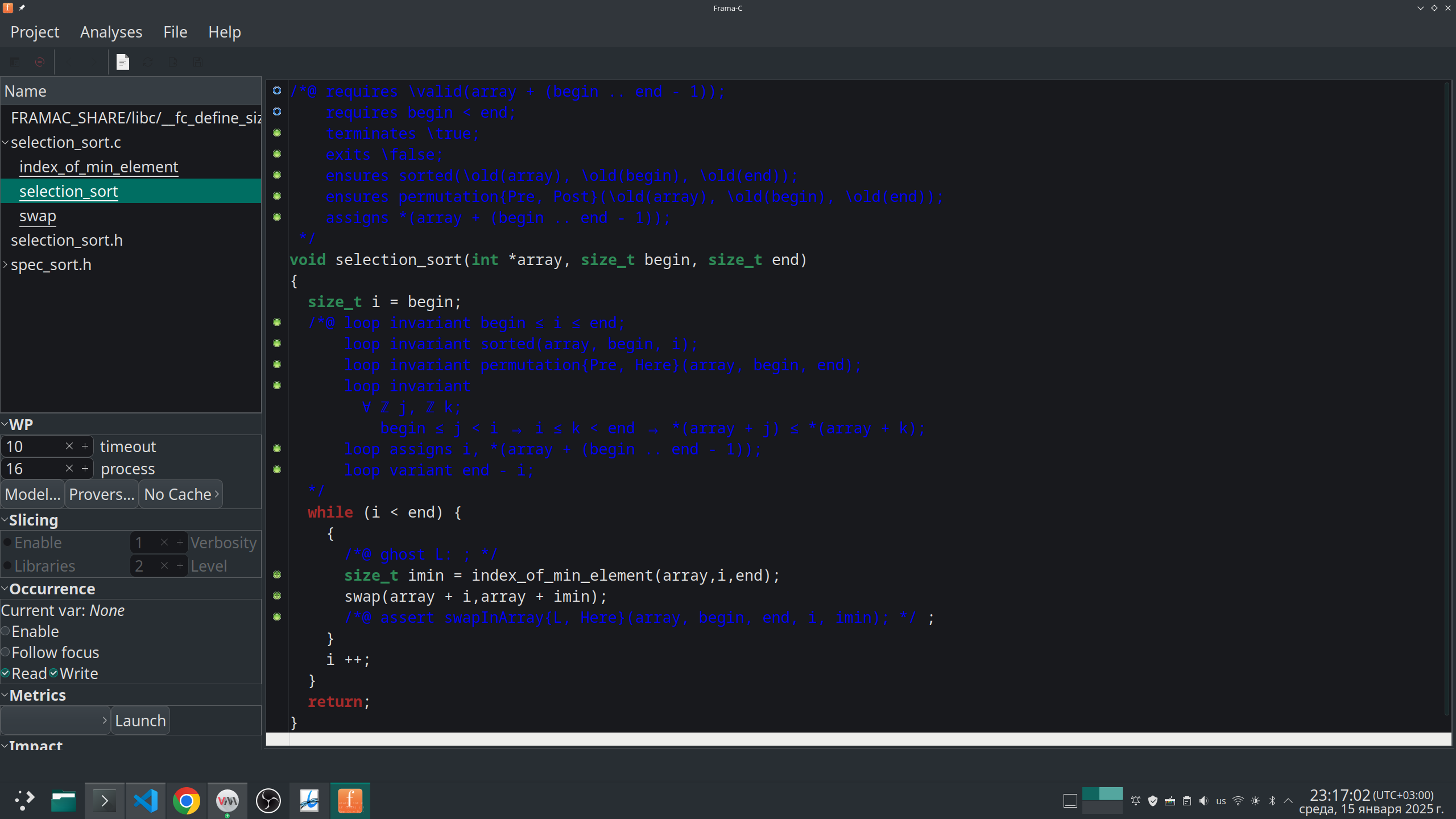
Данное постусловие оказалось вычислительно трудным для wp. Однако, ACSL предоставляет возможность помочь модулю дедуктивной верификации, введя опорные выражения в код функции с помощью ключевого слова *assert*. Добавим необходимые выражения (Листинг 2.11).

*Листинг 2.11 – Спецификация цикла функции selection\_sort с вспомогательными конструкциями*



Вспомогательное выражение говорит о том, что итерация цикла выполняет операцию обмена значений между двумя элементами массива. А так как данная операция является базой индукции в определении предиката *permutatuion*, опора на него позволяет wp доказать соответствующее условие в инварианте цикла.

Проверим, насколько теперь успешна верификация в Frama-C (Рисунок 2.6).



*Рисунок 2.6 – Успешная верификация постусловия сохранения элементов массива функции selection\_sort*

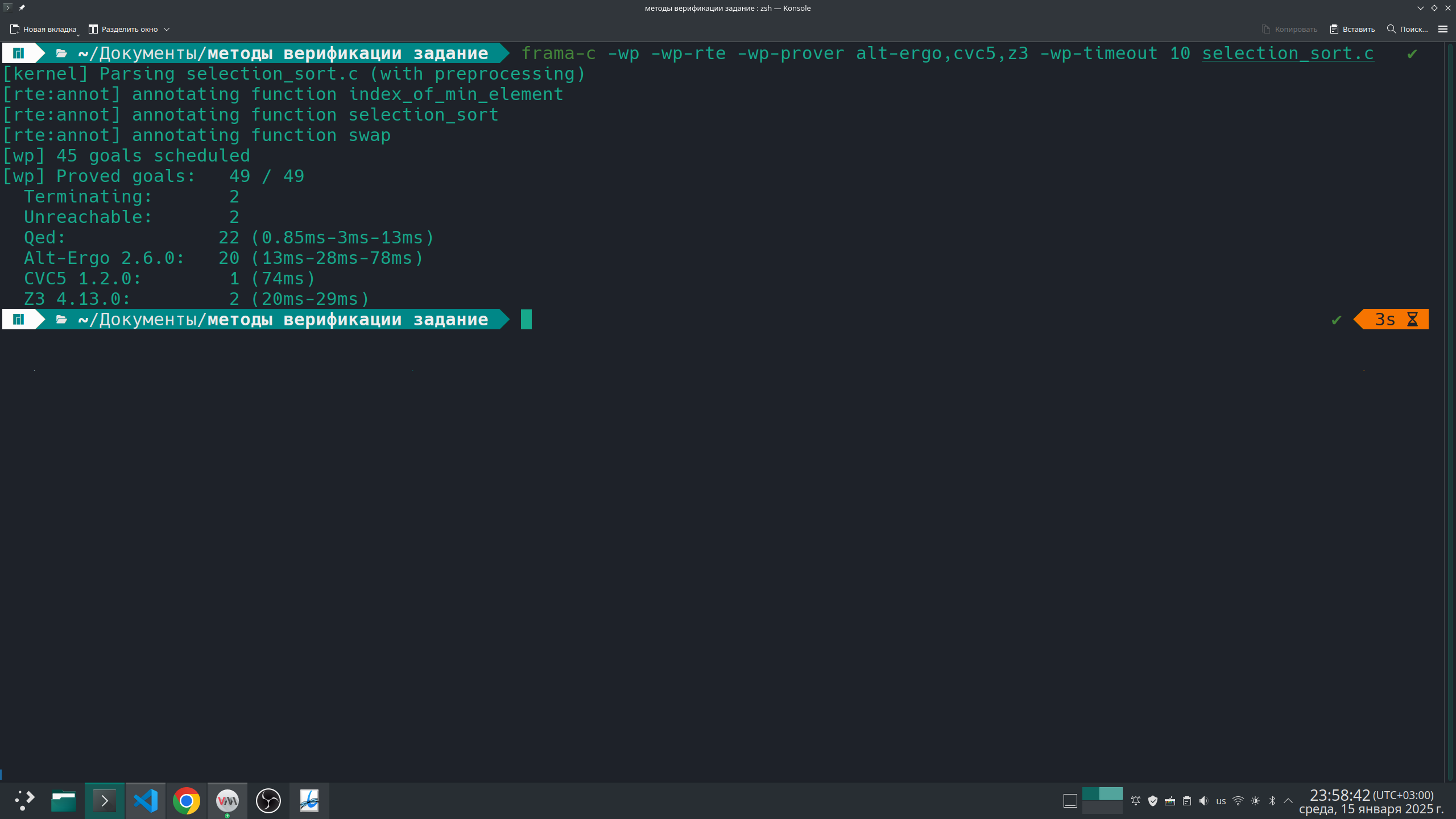
В введённых вспомогательных выражениях содержится такой элемент, как ghost код. Ghost код – это код Си внутри ACSL спецификаций, который может читать значения из оригинального кода, но не может записывать их в него или каким-либо другим образом влиять на исполнение оригинального кода.

Ghost код часто используется для построения вспомогательных выражений, которые упрощают верификацию сложных свойств. Например, в рассмотренном контракте, он используется для добавления метки в код функции.

Кроме этого, ghost код оказывается полезным для обхода ограничений Frama-C. Например, так как Frama-C не поддерживает работу с динамической памятью, часто для верификации функции выполняющих эту работу вводят ghost переменную количества сделанных вызовов *malloc* и подобных функций.

## 2.4 РЕЗУЛЬТАТ

Теперь Frama-C полностью верифицирует модуль сортировки выбором (Рисунок 2.7).



*Рисунок 2.6 – Верификция модуля сортировки выбором*

Полный код модуля со спецификациями расположен в Приложении Б.

# СПИСОК ЛИТЕРАТУРЫ

1. Juliet C/C++ 1.3. [Электронный ресурс] Режим доступа: https://samate.nist.gov/SARD/test-suites/112 (Дата доступа 19.01.2025).
2. CWE-562: Return of Stack Variable Address. [Электронный ресурс] Режим доступа: https://cwe.mitre.org/data/definitions/562.html (Дата доступа 19.01.2025).
3. PVS-Studio теперь в Compiler Explorer. [Электронный ресурс] Режим доступа: https://habr.com/ru/companies/pvs-studio/articles/509808/ (Дата доступа 19.01.2025).
4. Unspecified behavior in C/C++ with Examples. [Электронный ресурс] Режим доступа: https://www.geeksforgeeks.org/unspecified-behavior-in-c-c-with-examples/ (Дата доступа 19.01.2025).
5. A. Blanchard. Introduction to C program proof with Frama-C and its WP plugin [Электронный ресурс] Режим доступа: https://github.com/AllanBlanchard/tutoriel\_wp (Дата доступа 19.01.2025).

# ПРИЛОЖЕНИЕ А

*Листинг А.1 – CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01.c*

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | /\*   \* @description Return of local buffer address to local buffer   \*   \* \*/    #include "std\_testcase.h"    /\* this REQUIRES secondary functions \*/    #ifndef OMITBAD    **static** **char** \*helperBad()  {  **char** charString[] = "helperBad string";        /\* FLAW: returning stack-allocated buffer \*/  **return** charString; /\* this may generate a warning -- it's on purpose \*/  }    **void** CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01\_bad()  {      printLine(helperBad());  }    #endif /\* OMITBAD \*/      #ifndef OMITGOOD    **static** **char** \*helperGood1()  {  **static** **char** charString[] = "helperGood1 string";        /\* FIX: don't return a stack-allocated buffer       \* you can use static (i.e., global) variables but this renders your       \* code, and all code that uses it, non-re-entrant and non-threadsafe,       \* and hence is not a complete solution.  We do it anyway       \*/  **return** charString;  }    **static** **void** good1()  {      printLine(helperGood1());  }    **void** CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01\_good()  {      good1();  }    #endif /\* OMITGOOD \*/    /\* Below is the main(). It is only used when building this testcase on   \* its own for testing or for building a binary to use in testing binary   \* analysis tools. It is not used when compiling all the testcases as one   \* application, which is how source code analysis tools are tested.   \*/    #ifdef INCLUDEMAIN    **int** main(**int** argc, **char** \* argv[])  {      /\* seed randomness \*/  **srand**( (unsigned)**time**(NULL) );    #ifndef OMITGOOD      printLine("Calling good()...");      CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01\_good();      printLine("Finished good()");  #endif /\* OMITGOOD \*/    #ifndef OMITBAD      printLine("Calling bad()...");      CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_buf\_01\_bad();      printLine("Finished bad()");  #endif /\* OMITBAD \*/    **return** 0;  }    #endif |

*Листинг А.2 – CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_local\_class\_member\_01.cpp*

|  |  |
| --- | --- |
| 001  002  003  004  005  006  007  008  009  010  011  012  013  014  015  016  017  018  019  020  021  022  023  024  025  026  027  028  029  030  031  032  033  034  035  036  037  038  039  040  041  042  043  044  045  046  047  048  049  050  051  052  053  054  055  056  057  058  059  060  061  062  063  064  065  066  067  068  069  070  071  072  073  074  075  076  077  078  079  080  081  082  083  084  085  086  087  088  089  090  091  092  093  094  095  096  097  098  099  100  101  102  103  104  105  106  107 | /\*   \* @description Return of local buffer address via stack-allocated object   \*   \* \*/    #include "std\_testcase.h"    /\* this REQUIRES secondary functions \*/    **namespace** CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_local\_class\_member\_01  {    **class** HelperClass  {  **private**:  **int** intArray[2];    **public**:          HelperClass()          {              intArray[0] = 0x41;              intArray[1] = 0x42;          }    **const** **int** \*getIntArray()          {  **return** intArray;          }  };    #ifndef OMITBAD    **static** **const** **int** \*helperBad()  {      HelperClass HelperClass;        /\* FLAW: returning stack-allocated buffer \*/  **return** HelperClass.getIntArray(); /\* this may generate a warning -- it's on purpose \*/  }    **void** bad()  {      printIntLine(\*helperBad());  }    #endif /\* OMITBAD \*/      #ifndef OMITGOOD    **static** **const** **int** \*helperGood1()  {  **static** HelperClass HelperClass;        /\* FIX: don't return a stack-allocated buffer       \* you can use static (i.e., global) variables but this renders your       \* code, and all code that uses it, non-re-entrant and non-threadsafe,       \* and hence is not a complete solution.  We do it anyway       \*/  **return** HelperClass.getIntArray();  }    **static** **void** good1()  {      printIntLine(\*helperGood1());  }    **void** good()  {      good1();  }    #endif /\* OMITGOOD \*/    };    /\* Below is the main(). It is only used when building this testcase on   \* its own for testing or for building a binary to use in testing binary   \* analysis tools. It is not used when compiling all the testcases as one   \* application, which is how source code analysis tools are tested.   \*/    #ifdef INCLUDEMAIN    **using** **namespace** CWE562\_Return\_of\_Stack\_Variable\_Address\_\_return\_local\_class\_member\_01; /\* so that we can use good and bad easily \*/    **int** main(**int** argc, **char** \* argv[])  {      /\* seed randomness \*/  **srand**( (unsigned)**time**(NULL) );    #ifndef OMITGOOD      printLine("Calling good()...");      good();      printLine("Finished good()");  #endif /\* OMITGOOD \*/    #ifndef OMITBAD      printLine("Calling bad()...");      bad();      printLine("Finished bad()");  #endif /\* OMITBAD \*/    **return** 0;  }    #endif |

# ПРИЛОЖЕНИЕ Б

*Листинг Б.1 – selection\_sort.c*

**

*Листинг Б.2 – selection\_sort.h*

**

*Листинг Б.3 – spec\_sort.h*

**