

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

[2 ВЕРИФИКАЦИЯ ФУНКЦИИ В FRAMA-C 3](#_Toc187871720)

[2.1 ВЕРИФИКАЦИЯ ФУНКЦИИ SWAP 3](#_Toc187871721)

[2.2 ВЕРИФИКАЦИЯ ФУНКЦИИ INDEX\_OF\_MIN\_ELEMENT 4](#_Toc187871722)

[2.2.1 Предусловия и постусловия функции index\_of\_min\_element 4](#_Toc187871723)

[2.2.2 Верификация цикла функции index\_of\_min\_element 6](#_Toc187871724)

[2.3 ВЕРИФИКАЦИЯ ФУНКЦИИ SELECTION\_SORT 8](#_Toc187871725)

[2.3.1 Верификация постусловия неубывания элементов массива 8](#_Toc187871726)

[2.3.2 Индуктивное определение перестановки элементов массива 10](#_Toc187871727)

[2.3.3 Верификация постусловия сохранения элементов массива 11](#_Toc187871728)

[2.3.4 Вспомогательные конструкции внутри блока цикла 13](#_Toc187871729)

[2.4 РЕЗУЛЬТАТ 15](#_Toc187871730)

[ПРИЛОЖЕНИЕ А 16](#_Toc187871731)

# 2 ВЕРИФИКАЦИЯ ФУНКЦИИ В FRAMA-C

Рассмотрим код функции сортировки выбором (Листинг 2.1).

*Листинг 2.1 – Сортировка выбором*



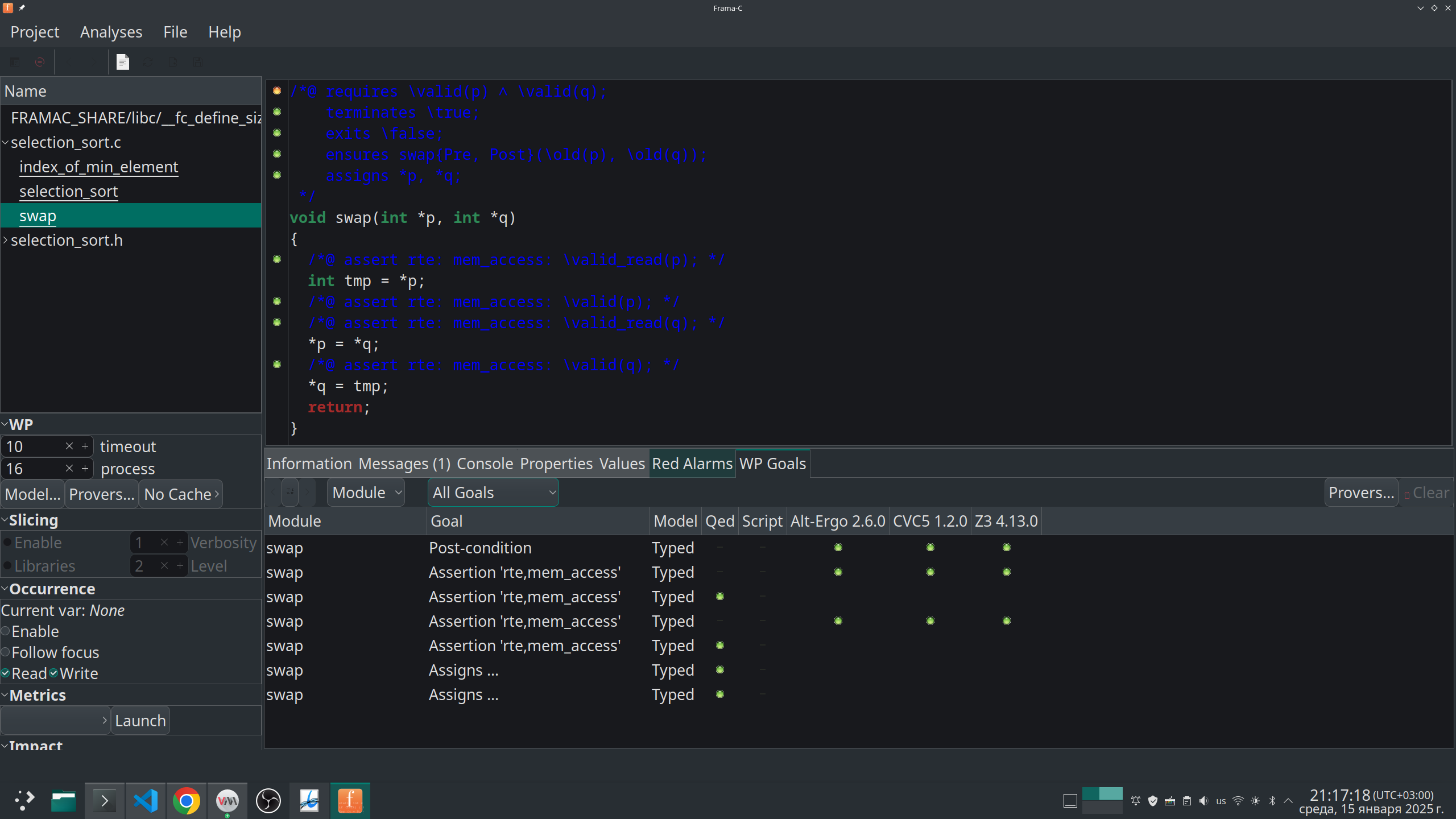
## 2.1 ВЕРИФИКАЦИЯ ФУНКЦИИ SWAP

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

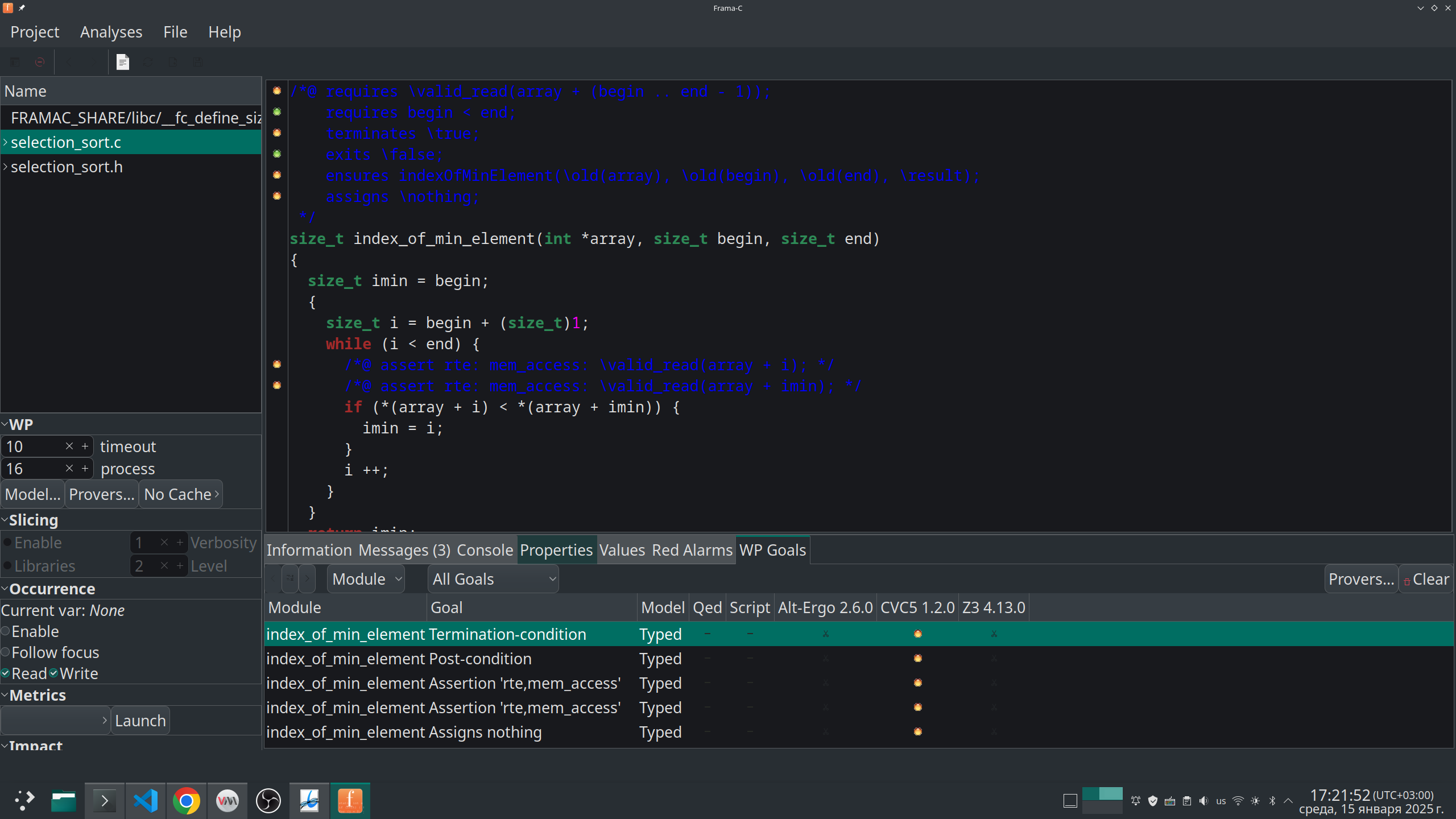
### 2.2.1 Предусловия и постусловия функции index\_of\_min\_element

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 инвариантой 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 вариантой цикла. Варианта задаёт верхнюю границу количества итераций цикла. Варианта обязательно должна быть неотрицательной в начале итерации, но может стать отрицательной 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*.

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*



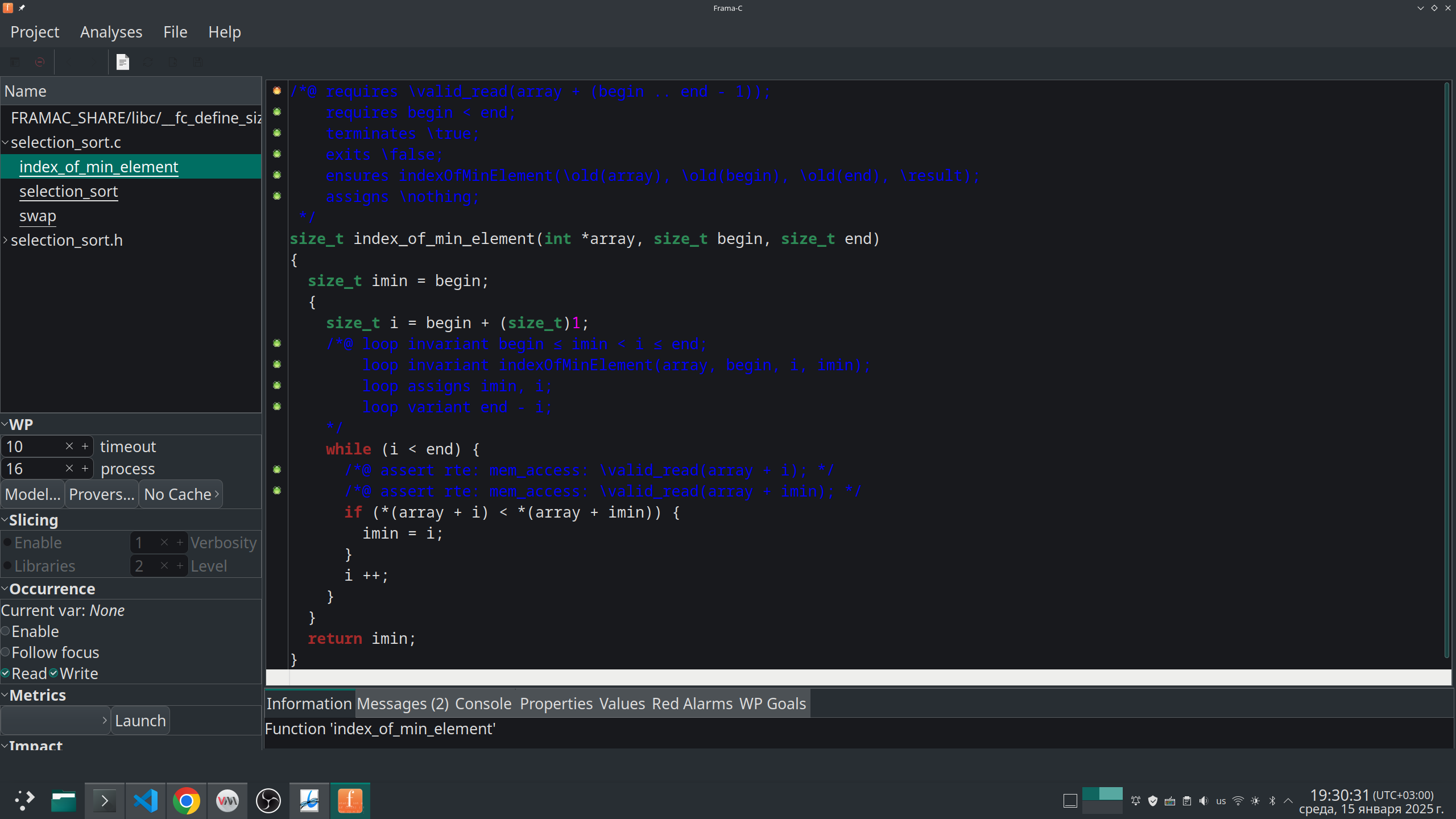
В данном случае первая часть инварианты задаёт количество раз, которое цикл изменяет переменные, вторая часть связывает инварианту с постусловием контракта функции, что необходимо для успешной её верификации.

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

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 неубывание 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*



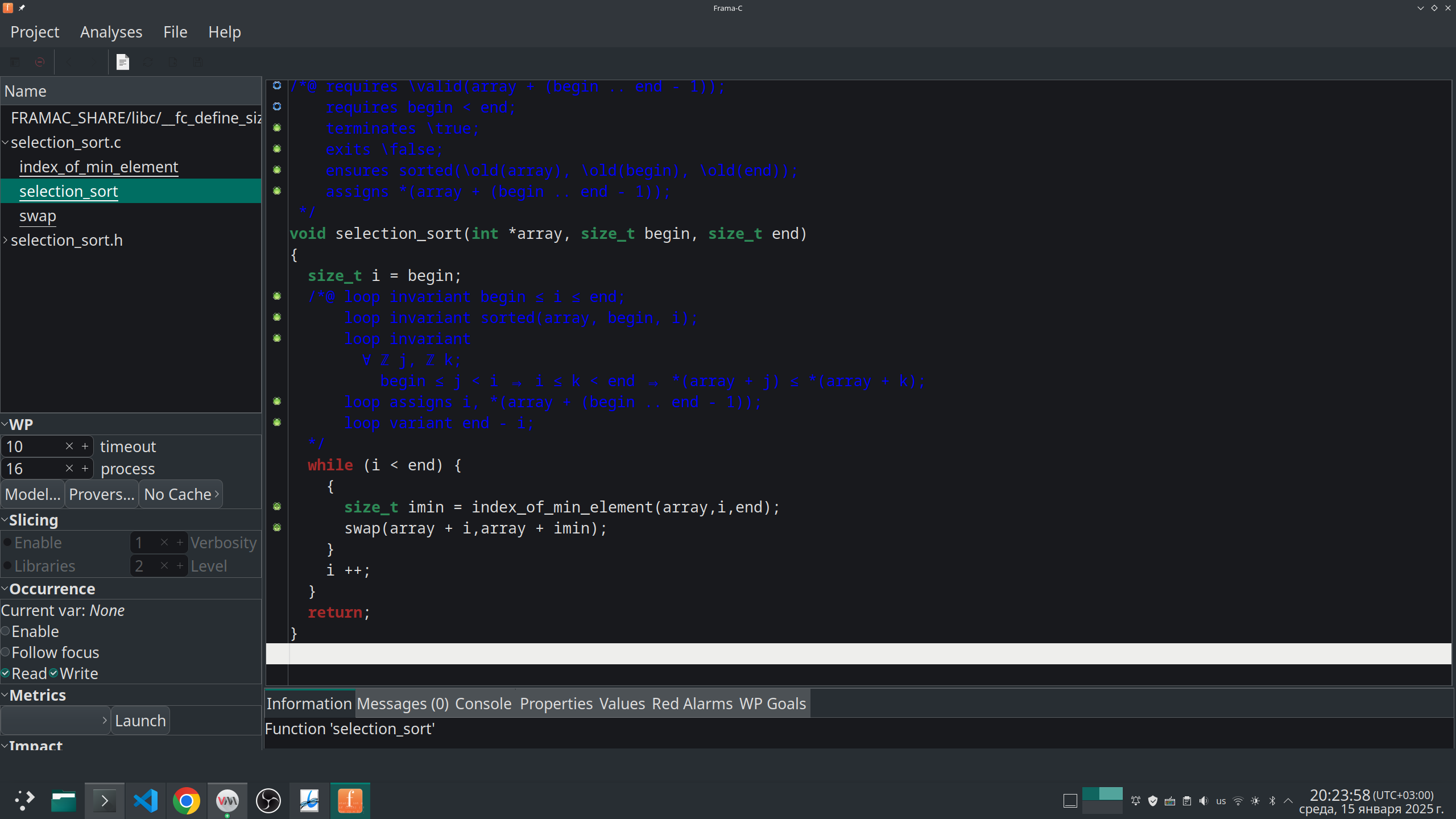
Не стоит забывать и о спецификации цикла (Листинг 2.6).

*Листинг 2.6 – Спецификация цикла функции selection\_sort с инвариантой неубывания 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 неубывания 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 ввести его индуктивно. 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*

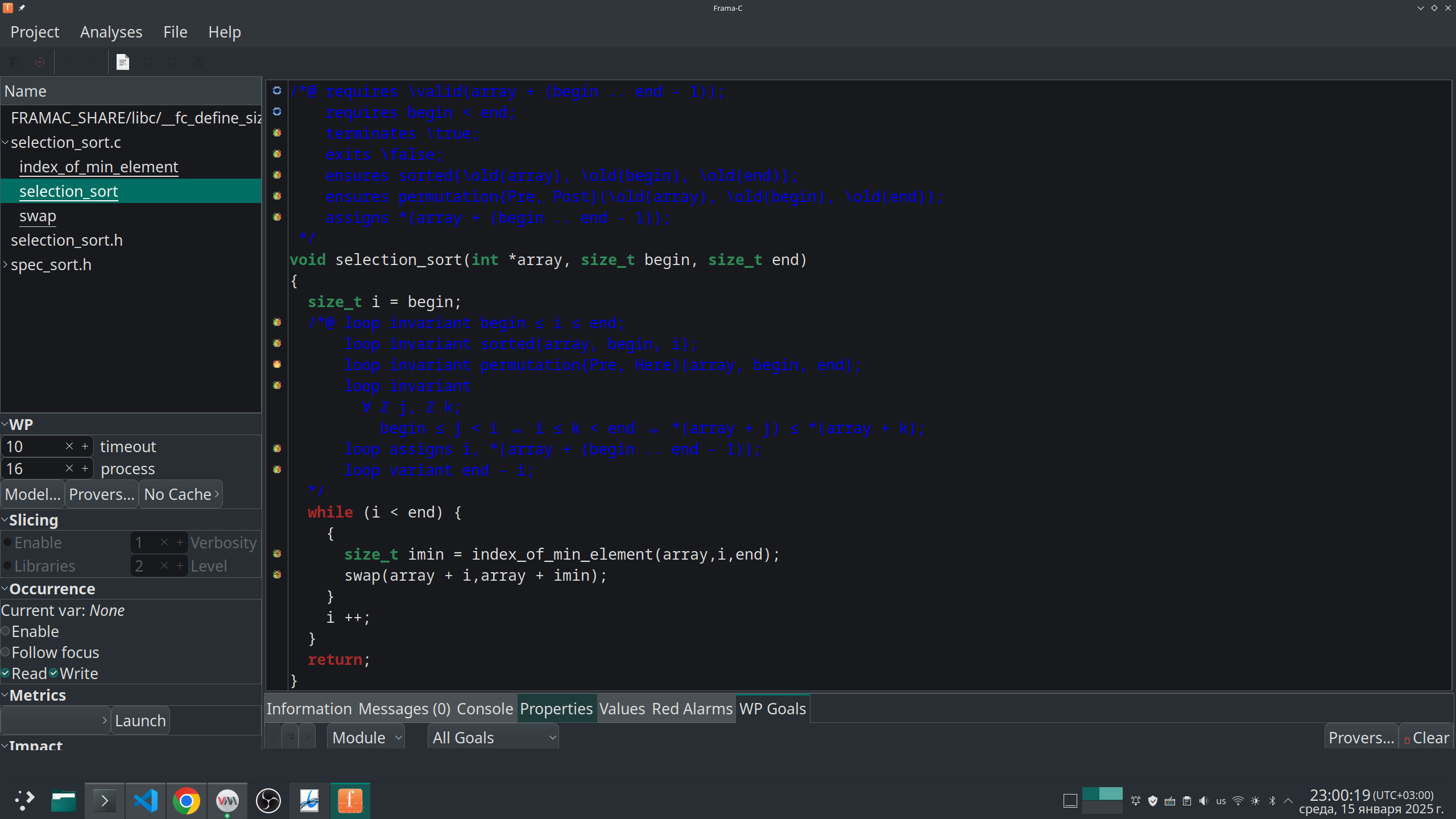
**

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

*Листинг 2.10 – Спецификация цикла функции selection\_sort с инвариантой 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

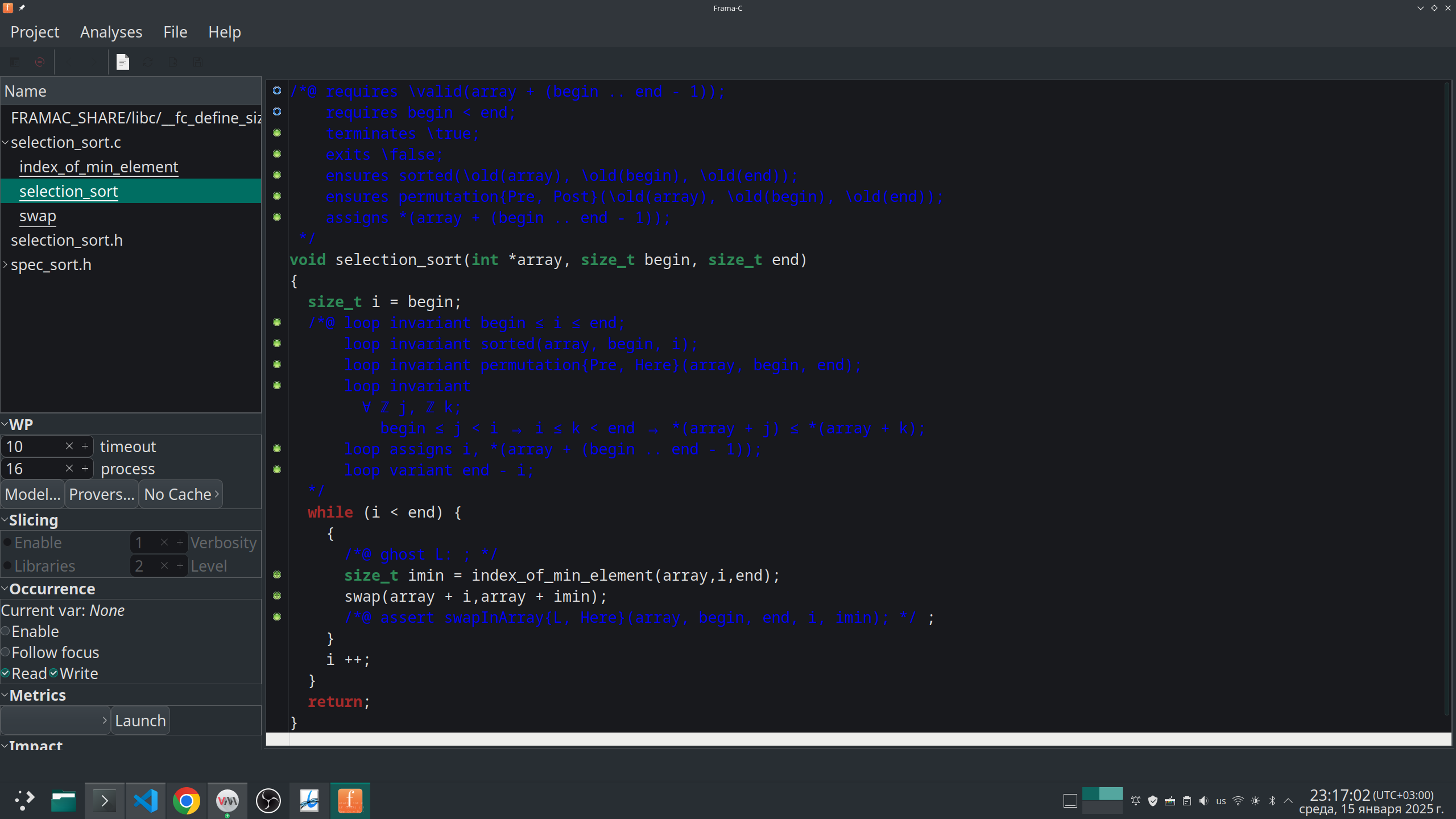
Данное постусловие оказалось вычислительно трудным для wp. Однако, ACSL предоставляет возможность помочь модулю дедуктивной верификации, введя опорные выражения в код функции с помощью ключевого слова *assert*. Добавим необходимые выражения (Листинг 2.11).

*Листинг 2.11 – Спецификация цикла функции selection\_sort с инвариантой сохранения элементов массива*



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*

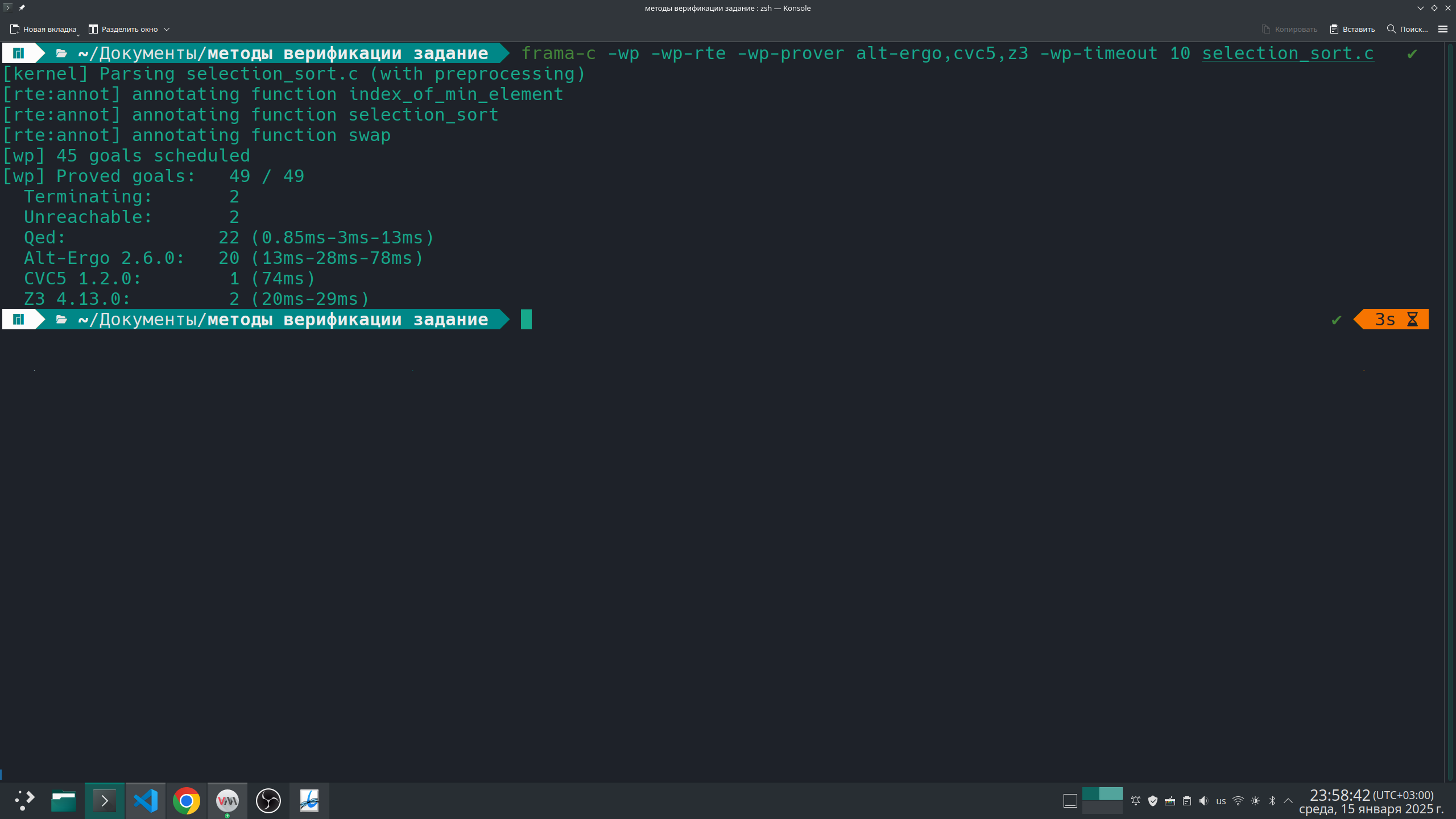
В введённых вспомогательных выражениях содержится такой элемент, как ghost код. Ghost код – это код Си внутри ACSL спецификаций, который может читать значения из оригинального кода, но не может записывать их в него или каким-либо другим образом влиять на исполнение оригинального кода.

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 А.

# ПРИЛОЖЕНИЕ А

*Листинг А.1 – selection\_sort.c*

**

*Листинг А.2 – selection\_sort.h*

**

*Листинг А.3 – spec\_sort.h*

**