Національний університет «Полтавська політехніка імені Юрія Кондратюка»

(повне найменування вищого навчального закладу)

\_Навчально науковий інститут інформаційних технологій та робототехніки\_

(повна назва факультету)

\_Кафедра комп’ютерних та інформаційних технологій і систем\_

(повна назва кафедри)

**Пояснювальна записка**

**до дипломного проєкту (роботи)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_бакалавра\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(освітньо-кваліфікаційний рівень)

на тему

Підвищення ефективності програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних інформаційно-управляючих систем

Виконав: студент 4 курсу, групи 402-ТК

спеціальності

\_\_123 Комп’ютерна інженерія \_

(шифр і назва напряму)

Данилейко А.С.

(прізвище та ініціали)

Керівник Васюта В.В.

(прізвище та ініціали)

Рецензент

(прізвище та ініціали)

Полтава – 2021 року

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**«ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»**

**НАВЧАЛЬНО НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ**

**КАФЕДРА КОМП’ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ**

**КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА**

**спеціальність 123 «Комп’ютерна інженерія»**

**на тему**

**«Підвищення ефективності програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних інформаційно-управляючих систем»**

**Студента групи 402-ТК Данилейка Артема Сергійовича**

Керівник роботи

кандидат технічних наук,

доцент Васюта В.В.

Консультант

кандидат технічних наук,

доцент Янко А.С.

Завідувач кафедри

кандидат технічних наук,

доцент Головко Г.В.

Полтава – 2021

**РЕФЕРАТ**

Кваліфікаційна робота бакалавра: 67с., 13 рисунків, 14 джерел.

**Об’єкт дослідження** – програмні засоби автоматизованої верифікації результатів компіляції програмних проєктів критичних інформаційно-управляючих (ІУС) систем українських АЕС.

**Предмет дослідження** – методики та процедури підвищення ефективності (поведінки в часі, утилізації ресурсів і т.д.) програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних ІУС.

**Мета дипломної роботи**: розробка програмних інструментів підвищення ефективності застосування програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних ІУС.

Для досягнення мети дипломної роботи поставлено такі **завдання** :

- провести аналіз процесу верифікації програмних проєктів критичних інформаційно-управляючих систем, обґрунтувати необхідність процесу його автоматизації;

- провести аналіз засобів автоматизації верифікації результатів компіляції платформи Radics;

- розробити пропозиції щодо підвищення ефективності роботи засобів автоматизації верифікації результатів компіляції платформи Radics;

- розробити програмну реалізацію щодо пропозицій і оцінити ефективність їх реалізації.

Дипломна робота складається з: вступу, трьох розділів, що включають 11 підрозділів, списку використаних джерел із 14 найменувань та висновку.

У тексті дипломної роботи міститься 13 рисунків. Загальний обсяг роботи 67 аркуша.

ЗМІСТ

[ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ 6](#_Toc74744600)

[ВСТУП 8](#_Toc74744601)

[1 КОНЦЕПТУАЛЬНІ ПІДХОДИ ОРГАНІЗАЦІЇ ПРОЦЕСІВ ВЕРИФІКАЦІЇ ПРОГРАМНИХ ПРОЄКТІВ КРИТИЧНИХ ІНФОРМАЦІЙНО-УПРАВЛЯЮЧИХ СИСТЕМ НА ОСНОВІ ВИМОГ ДО ЇХ ЖИТТЄВОГО ЦИКЛУ 10](#_Toc74744602)

[1.1 Автоматизація АЕС у вигляді програмно-технічних комплексів (ПТК). 10](#_Toc74744603)

[1.2 Загальні принципи реалізації критичних інформаційно-управляючих систем (ІУС) на основі програмованих логічних контролерів (ПЛК). 11](#_Toc74744604)

[1.3 Аналіз процесів верифікації прикладної логіки користувача в ІУС на основі ПЛК. 16](#_Toc74744605)

[1.4 Огляд основних процедур верифікації прикладної логіки спецплатформи RadiCS 32](#_Toc74744606)

[2 АНАЛІЗ ЗАСОБІВ АВТОМАТИЗАЦІЇ ВЕРИФІКАЦІЇ РЕЗУЛЬТАТІВ КОМПІЛЯЦІЇ ПЛАТФОРМИ RADICS 39](#_Toc74744607)

[2.1 Програмні засоби верифікації результатів компіляції програмних проєктів критичних інформаційно-управляючих систем 39](#_Toc74744608)

[2.2 Визначення можливих напрямків удосконалення функціоналу ROVT. 45](#_Toc74744609)

[2.3 Пропозиції щодо підвищення ефективності функціоналу ROVT. 47](#_Toc74744610)

[3 ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ROVT ЗА РАХУНОК РЕАЛІЗАЦІЇ БАГАТОПОТОЧНОСТІ. 53](#_Toc74744611)

[3.1 Аналіз способів реалізації багатопоточної роботи прикладного програмного забезпечення. 53](#_Toc74744612)

[3.2 Реалізація багатопоточності на мові програмування C#. 56](#_Toc74744613)

[3.3 Імплементація вибраних засобів багатопоточності в ROVT. 59](#_Toc74744614)

[3.3 Оцінка ефективності реалізованих засобів підвищення ефективності ROVT. 63](#_Toc74744615)

[ВИСНОВКИ 65](#_Toc74744616)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 66](#_Toc74744617)

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

**AFB** – Функціональні блоки додатків

**AL** – Логіка додатків

**CLR** – Common Language Runtime

**CPU** – Central processing unit

**CR** – Code review

**CSV** – Comma-Separated Values

**ED** – Бібліотека функціональних блоків

**EEPROM** – Electrically erasable programmable read-only memory

**EPROM** – Erasable Programmable Read Only Memory

**FPGA** – Field-Programmable Gate Array

**HTML** – HyperText Markup Language

**HTTP** – HyperText Transfer Protocol

**HW** – HardWare

**ICS** – International Commission on Stratigraphy

**IDE** – Integrated development environment

**JSON** – JavaScript Object Notation

**LM** – Логічний модуль

**NRC** – Nuclear Regulatory Commission

**PL** – Логіка платформи

**PROM** – Programmable read-only memory

**REST** **API** – Representational State Transfer application programming interface

**ROM** – Read-only memory

**ROVT** – RPCT Outputs Verification Tool

**RPCT** – Radiy Platform Configuration Tool

**SCA** – Static code analysis

**SLC** – Життєвий цикл програмного забезпечення безпеки

**SMT** – Simultaneous multithreading

**SQL** – Structured Query Language

**SSD** – Solid-State Drive

**SW** – SoftWare

**TCP** – Transmission Control Protocol

**TI** – Texas Instruments

**TUN** – Tunning

**UAL** – User Application Logic

**UCTF** – UAL Controller Test Framework

**UPD** – User Datagram Protocol

**VHDL** – Very high speed integrated circuits

**АЕС** – Атомна електростанція

**БД** – База даних

**БЖ** – Блок живлення

**ІУС** – Інформаційно-управляючі системи

**МАГАТЕ** – Міжнародне агентство з атомної енергії

**МЕК** – Міжнародна електротехнічна комісія

**МП** – Мікропроцесори

**ПЗ** – Програмне забезпечення

**ПЛК** – Програмований логічний контролер

**ПТК** – Програмно-технічний комплекс

**СУБД** – Система управління баз даних

**ЦП** – Центральний процесор

# ВСТУП

В даний час створення високоякісного програмного забезпечення є однією з найважливіших завдань розвитку науки та виробництва. Від того наскільки вдало зроблено програмне забезпечення системи залежить в кінцевому результаті її життєздатність. Проте, в зв’язку з тим, що тільки деякі суттєві властивості програмного забезпечення можуть бути виміряні безпосередньо і оцінені кількісними показниками, забезпечення належного рівня його якості набуває першочергового значення.

Таким чином, однією з актуальних та важливих задач програмної інженерії є реалізація ефективної технології забезпечення необхідного рівня якості програмних систем. Вирішення цієї задачі є особливо важливим для бізнес критичних програмних систем .

**Об’єкт дослідження** – програмні засоби автоматизованої верифікації результатів компіляції програмних проєктів критичних інформаційно-управляючих систем (ІУС) українських АЕС.

**Предмет дослідження** – методики та процедури підвищення ефективності (поведінки в часі, утилізації ресурсів і т.д.) програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних ІУС.

**Мета дипломної роботи**: розробка програмних інструментів підвищення ефективності застосування програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних ІУС.

Для досягнення мети дипломної роботи поставлено такі **завдання** :

- провести аналіз процесу верифікації програмних проєктів критичних інформаційно-управляючих систем, обґрунтувати необхідність процесу його автоматизації;

- провести аналіз засобів автоматизації верифікації результатів компіляції платформи Radics;

- розробити пропозиції щодо підвищення ефективності роботи засобів автоматизації верифікації результатів компіляції платформи Radics;

- розробити програмну реалізацію по пропозиціях і оцінити ефективність їх реалізації.

Дипломна робота складається з: вступу, трьох розділів, що включають 11 підрозділів, списку використаних джерел із 12 найменувань.

У тексті дипломної роботи міститься 13 рисунків. Загальний обсяг роботи 72 аркуша.

# 1 КОНЦЕПТУАЛЬНІ ПІДХОДИ ОРГАНІЗАЦІЇ ПРОЦЕСІВ ВЕРИФІКАЦІЇ ПРОГРАМНИХ ПРОЄКТІВ КРИТИЧНИХ ІНФОРМАЦІЙНО-УПРАВЛЯЮЧИХ СИСТЕМ НА ОСНОВІ ВИМОГ ДО ЇХ ЖИТТЄВОГО ЦИКЛУ

## Автоматизація АЕС у вигляді програмно-технічних комплексів (ПТК).

Програмно-технічні комплекси відносяться до автоматизованих систем.

Автоматизована система – система, що складається з персоналу і комплексу засобів автоматизації його діяльності, що реалізує інформаційну технологію виконання встановлених функцій.

Програмно-технічний комплекс автоматичного регулювання з використанням платформи RadICS

ПТК АРП-РОП-ППЗ, призначений для застосування в якості технічної бази при реконструкції діючих і створення нових систем управління і захистів з реактором ВВЕР.

Платформа апаратного забезпечення системи і ПТК АРП-РОП-ППЗ розроблено відповідно до діючих правил і рекомендацій Міжнародної електротехнічної комісії (МЕК), Міжнародного агентства з атомної енергії (МАГАТЕ) і Національного науково-дослідного ради США.

Безперервна діагностика системи забезпечується інтегрованими діагностичними пристроями, які обслуговуються незалежно від робочої схеми безпеки і об'єднаного кодування.

ПТК виконує наступні основні функції:

* автоматичного регулювання нейтронної потужності реактора і / або тиску в головному паровому колекторі турбіни енергоблоку АЕС;
* розвантаження і обмеження потужності реактора на рівнях відповідають обсягу включеного в роботу основного технологічного обладнання енергоблоку АЕС;
* прискореного попереджувального захисту (швидкого розвантаження реактора на 40-50% потужності за 3-4 сек.) при несанкціонованих відключеннях обладнання.

Для підвищення надійності спрацьовування захистів, в ПТК АРП-РОП-ППЗ реалізовано три рівня формування вихідних сигналів на основі мажоритарної логіки «два з трьох».

Якщо того вимагає процедура ліцензування, система може бути розроблена в конфігурації 2-з-4 відповідно до параметрів якості та надійності.

При проєктуванні ПТК АРП-РОП-ППЗ спочатку закладені принципи поділу функцій комплексу на керуючі і захисні. Різні групи функцій реалізовані в окремих, гальванічно розділених між собою, субблока, що підвищує експлуатаційну надійність комплексу в цілому.

## 1.2 Загальні принципи реалізації критичних інформаційно-управляючих систем (ІУС) на основі програмованих логічних контролерів (ПЛК).

Одним із варіантів використання ПЛК є сфера промислової автоматизації різноманітних технологічних процесів на великих і малих підприємствах. Популярність контролерів зрозуміла. Їх застосування значно спрощує створення і експлуатацію як складних автоматизованих систем, так і окремих пристроїв, в тому числі - побутового призначення. ПЛК дозволяє скоротити етап розробки, спрощує процес монтажу і налагодження за рахунок стандартизації окремих апаратних і програмних компонентів, а також забезпечує підвищену надійність в процесі експлуатації, зручний ремонт і модернізацію при необхідності [3].

Прийнято вважати, що завдання створення прообразу сучасного ПЛК виникла в кінці 60-х років минулого століття. Зокрема, в 1968 році вона була сформульована керівними фахівцями General Motors. Тоді ця компанія намагалася знайти заміну для складної релейної системи управління. Відповідно до отриманого завдання на проєктування, нова система управління мала відповідати таким критеріям як:

* просте і зручне створення технологічних програм;
* можливість зміни робочої керуючої програми без втручання в саму систему;
* просте і недороге обслуговування;
* підвищена надійність при зниженій вартості, в порівнянні з подібними релейними системами.

Наступні розробки в General Motors, Allen-Bradley та інших компаніях привели до створення системи управління на базі мікроконтролерів, яка аналізувала вхідні сигнали від технологічних датчиків і управляла електроприводами виконавчих пристроїв.

Термін ПЛК (Programmable Logic Controller, PLC) згодом був визначений в стандартах EN 61131 (МЕК 61131). ПЛК - це уніфікована цифрова керуюча електронна система, спеціально розроблена для використання у виробничих умовах. ПЛК постійно контролює стан пристроїв введення та приймає рішення на основі користувальницької програми для управління станом вихідних пристроїв. Склад і принцип дії представлений на рисунку 1.1

Спрощене уявлення складу і принципу дії ПЛК добре демонструє рисунок 1. З нього видно, що ПЛК має три основні секції:

* вхідні;
* вихідну;
* центральну.



Рисунок. 1.1 – Склад і принцип дії ПЛК

ПЛК може складатися з:

* модуля центрального процесора (CPU);
* модуля аналогових виходів;
* модуля аналогових входів;
* модуля комунікацій;
* модуля дискретних виходів;
* модуля дискретних входів;
* модуля керування осями;
* модуля лічильників;
* спеціальних модулів;
* блоків пам'яті ROM, PROM, EPROM, EEPROM.

Є ще джерело живлення. Можливе підключення до ПЛК зовнішнього ПК для програмування і налагодження.

Центральна секція містить центральний процесор (ЦП), пам'ять і систему комунікацій. Вона виконує обробку даних, що приймаються від вхідних секцій даних, і передає результати обробки в вихідну секцію. Слід відразу зазначити, що в великих ПЛК, крім ЦП, що діє в режимі «ведучий», можуть бути додаткові «ведені» ПЛК зі своїми ЦП. Як ЦП невеликого ПЛК використовуються стандартні мікропроцесори (МП). Зазвичай 8- і 16-розрядні МП цілком справляються з усіма стандартними завданнями. Але, як зазначено в МЕК 61131, вибір конкретного МП все ж залежить від завдань, покладених на даний тип ПЛК.

Для передачі даних іншому ПЛК або для підключення до мереж передачі даних PROFIBUS, Industrial Ethernet, AS-Interface в розподілених системах управління сьогодні використовуються комунікаційні процесори, такі як DP83867IR виробництва Texas Instruments (TI).

Вхідна секція ПЛК забезпечує введення в центральну секцію стану перемикачів, датчиків і смарт-пристроїв. Через вихідну секцію ЦП управляє зовнішніми виконавчими пристроями, серед яких можуть бути електромагнітні пускачі моторів, джерела світла, клапани та смарт-пристрої.

Типи ПЛК

Сучасні ПЛК, що використовують інноваційні технології, далеко пішли від перших спрощених реалізацій промислового контролера, але закладені в систему управління універсальні принципи були стандартизовані і успішно розвиваються вже на базі новітніх технологій.

Найбільшими світовими виробниками ПЛК сьогодні є компанії Siemens AG, Allen-Bradley, Rockwell Automation, Schneider Electric, Omron . Крім них ПЛК випускають і багато інших виробників, включаючи російські компанії ТОВ КОНТАР, Овен, Сегнетікс, Fastwel Груп, група компаній Текон і інші.

За конструктивним виконанням ПЛК ділять на моноблочні і модульні. У корпусі моноблочного ПЛК поряд з ЦП, пам'яттю і блоком живлення розміщується фіксований набір входів / виходів. У модульних ПЛК використовують окремо встановлюються модулі входів / виходів. Згідно з вимогами МЕК 61131, їх тип і кількість можуть змінюватися в залежності від поставленого завдання і оновлюватися з плином часу. Подібні ПЛК можуть діяти в режимі «ведучого» і розширюватися «веденими» ПЛК через інтерфейс Ethernet.

Моноблочні функціонально завершені ПЛК можуть включати в себе невеликий дисплей і кнопки управління. Дисплей призначений для відображення поточних робочих параметрів і вводяться за допомогою кнопок команд робочих програм і технологічних установок. Більш складні ПЛК комбінуються з окремих функціональних модулів, спільно закріплюються на стандартній монтажній рейці. Залежно від кількості обслуговуваних входів і виходів, встановлюється необхідна кількість модулів введення і виведення.

Джерело живлення може бути вбудованим в основний блок ПЛК, але частіше виконаний у вигляді окремого блоку живлення (БЖ), що закріплюється поруч на стандартній рейці.

Первинним джерелом для БЖ найчастіше служить промислова мережа 24/48/110/220/400 В, 50 Гц. Інші моделі БЖ можуть використовувати в якості первинного джерела постійної напруги на 24/48/125 В. Стандартними для промислового обладнання та ПЛК є вихідні напруги БЖ: 12, 24 і 48 В. У системах підвищеної надійності можлива установка двох спеціальних резервованих БЖ для дублювання електроживлення .

Для збереження інформації при аварійних відключеннях мережі електроживлення в ПЛК використовують додаткову батарею.

Як відомо, початкова концепція програмованого логічного контролера сформувалася за часів переходу з релейно-транзисторних систем управління промисловим обладнанням на що з'явилися тоді мікроконтролери. Подібні ПЛК з 8- і 16-розрядних МП обмеженою продуктивності досі успішно експлуатуються і знаходять нові сфери застосування.

Величезний прогрес у розвитку мікроелектроніки затронув всю елементну базу ПЛК. У них значно розширився діапазон функціональних можливостей. Кілька років тому немислимі були аналогова обробка, візуалізація технологічних процесів або навіть спільне використання ресурсів ЦП в якості безпосереднього керуючого пристрою. В даний час підтримка цих функцій входить в базову версію багатьох ПЛК.

Будова та принцип роботи ПЛК

Основним принципом дії ПЛК є циклічна робота, у якій контролер виконує по черзі окремі команди у такій послідовності, в якій вони записані у програмі. На початку кожного циклу програма зчитує стан входів контролера та записує їх. Після виконання всіх команд і визначення підрахунку актуального для даної ситуації стану виходів, контролер вписує стани виходів до пам'яті, що є таблицею стану виходів процесу, а операційна система виставляє відповідні сигнали на виходи, котрі керують виконавчими механізмами. Отже, всі сигнальні комбінації подаються у вхідний модуль контролера, а програма відслідковує їх картину та реагує зміною станів виходів на основі закладеного алгоритму.

Цикл роботи ПЛК може виглядати як послідовність кроків:

* Автодіагностика.
* Зчитування входів.
* Виконання програми.
* Комунікаційні завдання.
* Встановлення станів виходів.

## 1.3 Аналіз процесів верифікації прикладної логіки користувача в ІУС на основі ПЛК.

Надійність та безпека критично важливих систем управління інформацією залежать в основному від якості програмного забезпечення, за допомогою якого виконуються критичні функції. Приховані помилки (помилки, які не були виявлені під час тестування та перевірки) у критичному програмному забезпеченні є факторами ризику відмови системи. Незалежний огляд критичного програмного забезпечення, який підтверджує виконання заявлених функцій та забезпечує оцінку ймовірності прихованих помилок, є необхідною умовою регуляторних вимог для різних галузей.

З цієї точки зору, основними проблемами є: надійність незалежної перевірки, оцінка надійності прихованих дефектів, повноти тестового покриття

для критичного програмного забезпечення і, отже, кількісної оцінки функціональної безпеки.

Незалежна перевірка та перевірка є ключовим методом для кваліфікації критичного програмного забезпечення. Впровадження є обов'язковою законодавчою вимогою у таких критичних сферах, як ядерна енергетика ("Програмне забезпечення для комп'ютеризованих систем, важливих для безпеки на атомних електростанціях" - Серія стандартів безпеки МАГАТЕ) та космічній галузі (ECSS-Q-40B, ECSS-Q-80B) ) та інші.

Незалежний огляд в рамках кваліфікаційних випробувань програмного забезпечення ICS визначально визначає фактичні можливості для гарантування необхідного рівня безпеки та якості ICS для критичних програм у цілому.

Робота ISO базується на процедурах, відомих як ISO / IEC. Переклад з англійської "Verification" дає йому чітке тлумачення: Verification – Verification [6]. Щоб допомогти вам це зрозуміти, ви можете навести приклад типової перевірки: тестування програми або тестового пристрою. Випробування проводяться відповідно до певних вимог щодо предмета, що перевіряється, і реєструється, що вимоги виконуються. Результатом перевірки є відповідь на питання "Чи відповідає об'єкт вимогам?"

Рівні перевірки.

Етап 1. Аналіз об'єкта перевірки.

1.1. Визначення категорії безпеки функцій, що виконуються об'єктом верифікації.

1.2. Визначення властивостей об’єкта доказування, що впливають на обсяг доказів.

Етап 2. Огляд

2.1. Визначення ступеня незалежності, кваліфікації та відповідальності осіб, які проводять перевірку

2.2. Визначення, розробка та затвердження процедур верифікації з визначенням критеріїв аналізу, використання відповідних пристроїв та необхідних документів: технічні специфікації, програми та процедури випробувань, інструкції тощо.

2.3. Визначте коригувальні дії щодо недоліків, виявлених у процесі перевірки.

2.4. Визначте заходи щодо документування процесу перевірки.

Етап 3. Огляд.

3.1. Виконання перевірочних завдань: проведення необхідних спостережень.

Методи та процедури. Умови, за яких проводяться випробування, можуть бути реальними або змодельовані в лабораторних умовах (наприклад, перевірка на різних лабораторних стендах) для імітації реальних робочих умов.

3.2. Усунути недоліки, виявлені в процесі розгляду.

3.3. Документація вжитих дій, включаючи коригувальні дії:

Реєстрація та представлення результатів спостереження у необхідній формі,

Контроль, вимірювання, випробування тощо (тобто здійснювана діяльність.

Визначення придатності, доцільності та ефективності цієї діяльності (об’єкта) для досягнення поставлених цілей. Як правило, це журнали приймально-здавальних випробувань, періодичних дій, випробувань типу, випробувань на надійність, перевірочних робіт та приймання конструкцій, побудованої документації тощо.

3.4. Висновок за результатами перевірки відповідності програмних систем вимогам.

Рівень 4. Висновок за результатами перевірки відповідності об'єкта перевірки вимогам.

4.1. Аналіз результатів: аналіз отриманих даних, оцінка їх об'єктивності, повноти та достатності для прийняття рішення щодо об'єкта використання, що перевіряється, та його використання за призначенням.

4.2. Рішення про придатність випробуваного зразка для передбачуваного застосування та використання та підготовки документів, що засвідчують прийняття продукції.

Перевірка дозволяє своєчасно вносити виправлення та попередження.

Визначено заходи щодо усунення невідповідностей, щоб уникнути або мінімізувати претензії зовнішніх та внутрішніх споживачів, поліпшити умови експлуатації та використання тестового елемента.

Приховані помилки критичного програмного забезпечення є основним фактором ризику надзвичайних ситуацій для всієї системи. Незалежна перевірка критичного програмного забезпечення є обов'язковою вимогою міжнародної нормативної бази.

Мета верифікації - забезпечити перевірку та реалізацію об’єкта (запиту чи програмного коду) без несподіваних функцій.

Існує два основних методи перевірки та аналізу систем у процесах перевірки та тестування: перевірка та автоматичний аналіз - це статичні методи, які можна виконувати на всіх етапах процесу розробки системи, а тестування - це динамічний метод, який виконується, коли програма вже створена на місці, тобто на етапі впровадження системи та після її завершення.

Тестування – це процес запуску програми (або частини програми) з метою (або метою) пошуку помилок. Налагодження не є видом тестування. Хоча слова "налагодження" та "тестування" часто використовуються як взаємозамінні, вони означають різні види діяльності. Тестування – це діяльність, спрямована на виявлення помилок. Метою коригування є визначити точний характер відомої помилки, а потім виправити цю помилку. Ці два заходи пов’язані – результати тестування є вхідними даними для налагодження. Налагодження є складним процесом і має такі причини: Програмісту потрібні глибокі знання особливостей управління використовуваним обладнанням, операційною системою, середовищем та мовою програмування, реалізовані процеси, тип та характеристики різних помилок, методи налагодження та пов'язані з ними програмні засоби. Складність налагодження також можна простежити за взаємним впливом помилок у різних частинах програми. Виявлену програмістом помилку виправляють, після чого проводять новий тест, оскільки не можна виключати виникнення нових помилок під час процесу виправлення. Повне повторне тестування займає багато часу і, отже, дорого. Тому, система розділена на окремі частини і перевіряється лише та частина та інші пов’язані частини, де виявлено несправність. Перевірка систем, що має важливе значення для безпеки, має багато спільного з випробуванням систем із високими вимогами до надійності. Для виявлення найбільшої кількості дефектів слід застосовувати велике тестування, а для оцінки безпеки слід використовувати методи статичного тестування. Однак через дуже низький рівень відмов багатьох CS не завжди можливо кількісно визначити надійність за допомогою статичних тестів, оскільки для цього потрібна дуже велика кількість тестів. Ці тести лише дають підставу вважати той чи інший контроль якості безпечним.

Усі докази безпеки системи базуються на наступному припущенні: Кількість несправностей в системі, що призводять до надзвичайних ситуацій, набагато менше загальної кількості несправностей системи. Безпека повинна бути зосереджена на виявленні потенційно небезпечних помилок. Якщо виявлено, що ці помилки не виявляються або не виникають, але не мають серйозних наслідків, система вважається надійною. Доведення правильності програм було запропоновано як метод перевірки програмного забезпечення більше 25 років тому. Однак ці методи в основному застосовуються лише в лабораторіях. Практичні проблеми виготовлення сертифікату відповідності програмного забезпечення настільки складні, що деяким компаніям це невиправдано дорого використовувати ці методи при розробці традиційних систем. Але, як зазначалося раніше, для ряду КС економічно вигідно використовувати докази правильності системи, ніж усувати наслідки помилок. Незважаючи на те, що не є економічно вигідним надати докази правильності для більшості систем, іноді необхідно розробити докази безпеки, щоб продемонструвати, що програма відповідає вимогам безпеки. Підтвердження того, що програма відповідає специфікації, не вимагається для випадку безпеки. Потрібно лише довести, що виконання програми не призводить до помилок з небезпечними наслідками. Доказ відповідності цієї програми вимогам безпеки. Демонструючи безпеку, немає необхідності доводити, що програма відповідає специфікації. Потрібно лише показати, що виконання програми не призводить до помилок з небезпечними наслідками. Доказ відповідності цієї програми вимогам безпеки

Необхідними передумовами високого рівня надійності та безпеки критичного програмного забезпечення є наявність належного (ефективного) нормативно-методичного забезпечення, а також широкомасштабне використання інструментів для підтримки процесів кваліфікаційних випробувань (експертизи), що відображає сучасна динаміка стандартизації в галузі розвитку інформаційних технологій та програмного забезпечення. Основним напрямком підвищення надійності оцінки якості програмного забезпечення ICS для критичного використання в кваліфікаційних випробуваннях є диверсифікація технологій верифікації.

Виконання цих умов визначально визначає реальні можливості гарантувати необхідний рівень безпеки та якості всієї ІСУ, також у рамках підходів до регулювання безпеки з урахуванням ризиків.

Якість програмного забезпечення ICS для критичних програм, враховуючи ризики прихованих програмних помилок, є невідкладною частиною впровадження підходів до регулювання безпеки та кваліфікації ICS для критичних програм у різних сферах застосування (атомні електростанції, космос, транспорт тощо).

Першим етапом огляду є здійснення статичного аналізу коду (SCA) та розгляд коду (огляд коду - CR) мовою SW. Цей код може описувати електронну конструкцію (ED) або його функціонально завершені компоненти, які призначені для конфігурації інтегральних схем FPGA, CPLD, ASIC, наприклад Бібліотека функціональних блоків (далі - ED). Метод SCA та CR для SW-коду (далі – SW-SCA / CR) використовується для перевірки відповідності аналізованого коду правилам кодування.

Процес SW SCA / CR використовується для перевірки відсутності помилок кодування та забезпечення ефективності розробки електронних конструкцій.

Схема алгоритму для проведення статичного аналізу коду (SCA) та для врахування коду (огляд коду) показана на рис. 1. 1.1.

Процедура SCA / CR-SW може бути застосована до ED, для якого завершено етап кодування [2].

Склад незалежної групи з верифікації для проведення SCA / CR-SW.

Процес SW SCA / CR здійснюється незалежною групою фахівців з верифікації (далі - група), які не брали участь у розробці IDE, які мають необхідні знання, навички та досвід у розробці (синтезі) та перегляду коду функціонально-складних IDE.

1.1. Статичний аналіз коду та схема алгоритму перегляду коду.

Керівник групи перевірки відповідає за:

- Формування складу групи для SW SWA / CR;

- Планування, підготовка та впровадження SW SCA / CR;

- Гарантоване виконання цієї процедури у правильному порядку;

- Створення звітних документів за результатами SW SCA / CR.

Член групи верифікації (далі - верифікатор) відповідає за запуск SW SCA / CR, щоб перевірити відсутність недоліків. Аудитор може виступати в ролі читача (при виконанні SW-CR) та / або як реєстратора (при виконанні SW-SCA / CR). Менеджер може також виступати в ролі аудитора. Роль перевіряючого не може бути призначена працівникам, які брали участь у розробці. Перевіряється відповідність коду ED. Для запуску програмного забезпечення SCA / CR рекомендується призначити принаймні двох рецензентів.

Кількість рецензентів можна збільшити, якщо код IDE є особливо складним.

Повинні бути визначені обов'язки верифікатора:

- читач - це головна особа, відповідальна за читання матеріалів, що надаються та підпадають під дію SW CR; Його мета - привернути увагу до матеріалів, що надаються SW CR, та забезпечити впорядкований потік інформації для сприйняття іншими рецензентами;

- Реєстратор призначений для запису даних про помилки, невідповідність даних про SW SCA / CR.

До SW SCA / CR також повинен бути залучений розробник - фахівець із групи розробки ED, який відповідає за створення та усунення виявлених помилок та / або невідповідностей. Під час SW SCA / CR він відповідає за попереднє подання проєкту IDE до тестових центрів.

Передача коду ED розробником команді верифікації SW SCA / CR.

Код ED генерується в середовищі проєктування (інструменті) для певного типу FPGA. Сертифікат (ліцензії) повинен бути доступним для такого інструменту (ІС), і, якщо це можливо, затвердження ІС, яке проводиться при виборі ІС для конструкції IDE, не є частиною описаних процесів SW SCA / CR у цій процедурі. Функціональний пакет даних про безпеку, який був сертифікований незалежним органом сертифікації – TÜV Rheinland Group.

Після створення (синтезу, розробки), попереднього моделювання функціонального тесту та налагодження зазначеного коду ED, наступне передається керівнику групи верифікації для прямого SW SCA / CR у зазначеному порядку за рішенням керівник групи з розвитку IDE:

- повний набір дійсних технічних проєктних документів, які будуть використовуватися як вхідні дані для розробки коду IDE (включаючи специфікації для розробки продукту, концепцію товару, опис архітектури товару);

- перелік та короткий опис функціональних бібліотек плагінів розробниками коду ED в середовищі проєктування ED та опис процедури підключення (роз'єднання) цих бібліотек при розробці коду, що підлягає SW SCA;

- Повна електронна копія функціональних бібліотек плагінів, що використовуються розробниками коду ED в середовищі дизайну ED при розробці коду, який підлягає SW SCA.

Попереднє подання проекту ED (наприклад, подання опису архітектури (AD) або детального опису (DD)) є необов’язковим кроком у процедурі SCA / CR-SW, коли розробник подає проєкт ED, повідомляє SCA / верифікатори . CR SW. Керівник групи з верифікації відповідає за визначення потреб та планування цього етапу.

Попереднє подання проєкту IDE має здійснюватися з огляду на великий розмір та складність проєкту IDE та недостатні знання рецензентів.

Попереднє подання проєкту IDE завжди повинно здійснюватися (включаючи повторне подання) кожного разу, коли створюються нові документи або вносяться зміни до оригінальних документів, необхідних для SW SCA / CR.

Критеріями вступу для попереднього подання проєкту IDE є:

- Про SCA / CR-SW верифікаторів повідомляє керівник групи з верифікації.

- Керівник групи з верифікації визначив необхідність попереднього подання проєкту IDE.

- Команда розробників IDE готова надати матеріали для попередньої презентації проєкту IDE.

Завдання, які слід вирішити на етапі попереднього подання проєкту IDE:

1. Керівник групи з верифікації повідомляє верифікаторів про майбутню зустріч з попереднього подання проєкту IDE.

2. Попередня презентація одним із фахівців групи розробки IDE проєкту IDE.

3. Переконайтеся, що вимоги етапів підготовки та впровадження ПЗ SCA / CR відповідають цілям брифінгу з питань IDE (для підтвердження рецензентами).

Початковим критерієм для цього рівня SW SCA / CR є схвалення аудиторами для задовільного проведення наради при попередньому поданні проєкту IDE у формі протоколу засідання, складеного керівником аудиторської групи.

Отримавши всі супровідні та описові документи для розробленого коду IDE (у паперовій та / або електронній формі) та повні, точні електронні копії розробленого коду IDE, затверджені відповідними посадовими особами, і все знаходиться в розробці з дизайнерським середовищем, пов’язаним кодом середовища функціональних бібліотек (в електронній формі) керівник групи верифікації виконує детальний план у суворій відповідності із загальним планом верифікації та валідації та розподіляє ролі та відповідальність за безпосередню реалізацію SW SCA / CR в команді членів.

Планування складається з ідентифікації підрозділів, що лежать в основі SCA / CR SW, та визначення часу, необхідного для виконання SCA / CR SW. У розкладі SCA / CR-SW повинні бути вказані дата та час проведення SCA / CR-SW та склад групи з верифікації.

Критерії вступу для планування SW SCA / CR:

- Викладено вимоги до плану SW SCA / CR.

- За попередніми оцінками, встановлений розмір коду IDE;

- Екзаменатори закінчили професійне навчання або спеціальний план їхнього навчання;

- Визначте цілі та завдання для SW SCA / CR.

Критерії вступу для планування SW SCA / CR:

- Описова документація для розробленого коду IDE наближається до завершення.

- наявні ресурси необхідні для виконання SCA / CR SW;

- керівник групи з верифікації повідомляє верифікаторів SCA / CR SW.

Завдання, які слід вирішити на етапі планування SW SCA / CR:

1. Визначте об’єкт, якому підпорядковується SCA / CR SW та правила правильного застосування, код ED якого буде перевірятися.

2. Оцініть ресурси для виконання SCA / CR-SW та розподіліть часовий бюджет.

Для виконання SW CR та аналізу результатів SW SCA / CR рекомендується планувати час відповідно до кількості коду - 100-150 рядків коду на годину. Під час багаторазового виконання SW CR вищезазначені дані можуть бути виправлені керівником групи з верифікації.

3. Встановіть етапи для програмного забезпечення SCA / CR.

4. Визначте залежність від інших груп, які беруть участь у формуванні коду IDE або відповідної описової документації.

Завдання, які необхідно вирішити на етапі планування SW SCA / CR:

1. Визначте склад групи з верифікації SW SCA / CR та розподіліть обов'язки між верифікаторами.

2. Визначте дату рівнів SW SCA / CR.

3. Перевірте відповідність коду ED, поданого розробниками для SW CR, вимогам, зазначеним у Розділі 3 цієї процедури.

4. Провести порівняльний аналіз версій компонентів IDE, які підпадають під метод SCA / CR-SW у випадку фази регресії SCA / CR.

5. Сплануйте SW SCA / CR.

Етап планування завершується, коли виконуються наступні критерії:

- розроблено план SW SCA / CR;

- Якщо неможливо визначити точні дати деяких фаз плану, SW SCA / CR повинен вказати межі можливих дат.

- У плані SCA / CR-SW вказано достатньо ресурсів.

- складено графік SW;

- Обов'язки розподілені між аудиторами.

6. Метод статичного аналізу для коду IDE

Процес SW-SCA здійснюється з ІС SW-SCA, який вибирається як частина аналізу та вибору ІС для конкретного проєкту, а результат якого відображається у відповідному звіті. Ця процедура описує процедуру виконання SW-SCA з використанням наступних ІС:

7. Метод перевірки коду ED

Для того, щоб верифікатори могли виконати SW CR, керівник групи верифікації перевіряє наявність описової документації для розробленого коду ED та результатів SW SCA верифікаторам, які повинні підготуватися на цьому етапі проводиться засідання для аналізу результатів SW SCA / CR.

Основна відповідальність за впровадження SW CR лежить на верифікаторах.

Критеріями введення для верифікатора SW CR є:

- попереднє подання проєкту IDE, якщо це необхідно, було успішно завершено;

- описова документація на розроблений код IDE;

- Для проведення SW CR доступні необхідні матеріали-носії.

- Час, необхідний для заповнення SW CR, був погоджений з тестовими центрами.

Завдання, які необхідно вирішити при виконанні SW CR:

1. Проаналізуйте результати SW SCA, дослідивши порушення, виявлені ІС HD Designer.

2. Кожен екзаменатор повинен вказати, скільки часу потрібно для заповнення SW-CR.

Фаза запуску SW CR завершена, коли виконуються наступні критерії:

- Кожен екзаменатор заповнив лист-підтвердження SW CR.

- Підготовлені запитання, перелік невідповідностей та / або недоліків, виявлених у цьому Кодексі IDE, та коментарі до результатів SW SCA.

Аналіз результатів SW SCA / CR

Під час наради, на якій аналізуються результати SCA / CR-SW, обговорюються проблеми, аудитори виявляють невідповідності та / або недоліки та приймають рішення про ступінь важливості виявлених невідповідностей та / або недоліків. При обговоренні поданих матеріалів, які підпадають під дію SCA / CR SW, верифікатори можуть виявити нові невідповідності та / або недоліки, які не були виявлені під час SW CR. Варіанти усунення виявлених невідповідностей та / або помилок не повинні обговорюватися під час аналізу результатів SW SCA / CR.

Відповідальність за формування результатів впровадження SW SCA / CR покладається на всіх верифікаторів.

Критеріями введення для успішної зустрічі з розгляду SW CR є:

- Верифікатори, що виконують SW CR, повністю присутні;

- Для реалізації SW CR були доступні описові документи для розробленого коду ED.

- Визначення цілей наради для аналізу результатів впровадження SW SCA / CR.

Завдання, які потрібно вирішити під час проведення наради для аналізу результатів перегляду коду:

1. Короткий вступ керівника групи верифікації, в якому він повинен:

- забезпечити, щоб усі випробувальні центри, які виконують SW CR, усвідомлювали свою відповідальність;

- За необхідності проінформувати членів групи перевірки про структуру процедури SW SCA / CR (рис. 1) та місце SW CR у ній.

2. Перегляд рівня підготовки екзаменаторів керівником екзаменаційної групи, що складається з:

- опитування аудиторів щодо часу, який вони витратили на виконання SW CR, а також кількості та основних типів помилок та / або невідповідностей, виявлених під час аналізу;

- Прийняти рішення про те, що засідання з метою аналізу результатів впровадження SW CR базується на всіх раніше підготовлених коментарях та охоплює всі аспекти аналізу.

3. Прочитати список матеріалів, наданий читачем для SW CR.

4. Виявлення невідповідностей та / або недоліків аудиторами:

- виявляти невідповідності та / або дефекти за результатами виконання SW CR;

- Порівняти отримані результати SW SCA з результатами SW CR та виявити невідповідності та / або помилки для їх подальшого усунення (щодо пріоритетності (значущості) наслідків виявлених невідповідностей зазначених вимог щодо безпеки, які слід визначається);

- оцінити ступінь впливу невідповідностей та / або недоліків, які були виявлені під час впровадження SW SCA / CR на інші документи (організації), раніше створені в рамках відповідного проєкту, та визначити необхідність їх змін;

- Якщо аудитори вирішать, що невідповідності та / або недоліки, виявлені під час впровадження SW SCA / CR, призводять до зміни раніше створених документів (організацій) в рамках конкретного проєкту, що розглядається, керівник групи аудиторів ініціює зміну і обробляють згідно з описаним у внутрішніми інструкціями способом;

- Не обговорюйте стиль чи варіанти виправлення виявлених невідповідностей та / або недоліків.

5. Запис усіх розбіжностей та / або недоліків реєстратором, який повинен:

- вказувати будь-які виявлені невідповідності та / або дефекти, вказуючи їх місце в розглянутому коді IDE та номер порушеного правила;

- під час або після завершення засідання для аналізу результатів виконання ПЗ КЗ, перевірити виявлені невідповідності та / або недоліки, щоб потім подати їх аудиторам на затвердження;

- записати час, проведений на засіданні для аналізу результатів впровадження SW CR.

6. Визначити статус коду ED після прийняття одного з двох можливих рішень:

- якщо не виявлено невідповідностей та / або недоліків, результат SW CR затверджується як позитивний, а керівник групи верифікації готує спільний звіт для SW SCA / CR щодо проведеної процедури та досягнутих результатів;

- Якщо виявляються невідповідності та / або недоліки, приймається рішення про подальшу передачу коду IDE групі розробників IDE для виправлення, а потім організовується повторний SW SCA / CR і результати проміжного аналізу перетворюються у результати впровадження SW CR у формі проміжного звіту або проєктів підсумкових звітів.

Зустріч з огляду результатів впровадження SW CR завершується, коли виконуються наступні критерії:

- якщо код ED пройшов процедуру SW-CR у передбачуваному обсязі;

- були зафіксовані невідповідності та / або недоліки, їх місце розташування чітко вказано в цьому матеріалі та цей перелік затверджений рецензентами.

- Якщо виявляються невідповідності та / або недоліки, керівник групи верифікації подає код ED та перелік невідповідностей та / або недоліків на основі результатів SCA / CR-SW групі розробників коду ED для усунення.

Новий SW SCA / CR.

Критеріями вступу для оновленого впровадження SW SCA / CR є:

- Розробник завершив усунення невідповідностей та / або помилок у коді ED та подав SCA / CR-SW для повторного проведення.

- IDE-код із виправленими розбіжностями та / або помилками надіслано керівнику групи рецензування або зазначеному ним рецензенту.

Завдання, які слід вирішити при повторенні статичного аналізу коду та огляду коду:

1. Знову запустіть SW SCA

2. Знову проведіть SW CR

3. Керівник групи з верифікації готує загальну доповідь про результати SCA / CR-SW.

Фаза SCA / CR-Re-SW завершується, коли виконуються наступні критерії:

- Код ED, наданий для другого SW SCA / CR, пройшов цю процедуру.

- відхилень та / або дефектів не виявлено;

- Якщо були виявлені нові невідповідності та / або недоліки, їх потрібно надіслати знову до групи розробників коду ED для усунення (з подальшим поновленням виконання SCA / CR SW) (включаючи документальне пояснення причин окремих невідповідностей та / або недоліки неможливо усунути). Це рішення буде узгоджено між керівниками групи з верифікації та командою з розробки коду ED.

- Результати повторних SW SCA / CR фіксуються у загальному звіті про результати SW SCA / CR.

Для завершення SW SCA / CR потрібно:

- всі виявлені невідповідності та / або недоліки були усунені або причини, через які окремі невідповідності та / або недоліки не можуть бути усунені, задокументовані (рішення про прийнятність таких декларацій узгоджується між керівниками групи з верифікації та команда розробників коду ED і робиться разом).

- Керівник групи з верифікації заповнив та затвердив загальний звіт про результати SCA / CR-SW.

Огляд процедури SW SCA / CR - це фаза, на якій класифікуються причини недоліків, виявлених під час SW SCA / CR. Ця діяльність є важливим кроком у запобіганні появі подібних дефектів у майбутній роботі.

Керівник групи з верифікації планує зустріч за результатами процесу SCA / CR-SW. Усі екзаменатори беруть участь у цій зустрічі.

Завдання, які слід вирішити під час наради з урахуванням результатів процедури SW SCA / CR:

1. Виберіть усі помилки зі звіту SCA / CR SW для обговорення.

2. Визначте можливі причини помилки.

3. Запишіть результати цієї зустрічі. Також можуть бути зроблені рекомендації щодо вдосконалення існуючих процедур розробки та / або перегляду коду IDE, які потім представляються групі розробників коду IDE для подальшого запобігання помилкам.

Процедура реєстрації результатів SW SCA / CR.

Усі заходи щодо впровадження SW SCA / CR повинні бути детально задокументовані та можуть бути введені в систему відстеження помилок, щоб реєструвати та контролювати невідповідності та / або недоліки та відстежувати процес усунення цих невідповідностей та / або недоліків. Після завершення SWA SC / CR керівник групи з верифікації повинен підготувати звіт про документацію результатів за шаблоном, наведеним у Додатку А (виступ проєкту звіту залежить від вимог до конкретного проєкту IDE).

Звіт SCA / CR SW повинен містити:

- Дані про об'єкт SW SCA / CR;

- атрибути IS від SW SCA;

- Опис процедури реалізації SW SCA / CR;

- Порівняльний аналіз версій компонентів IDE, які підпадають під процедуру SW SCA / CR (підготовлений як додаток до основного звіту. Приклад можна знайти в Додатку Е).

- первинні та проміжні результати SW SWA / CR (включаючи порушення правил, коментарі, рекомендації);

- результати оцінки впливу невідповідностей та / або недоліків, виявлених під час виконання SW SCA / CR, на інші документи (компанії), які раніше були розроблені в рамках відповідного проекту, що розглядається (зазначити необхідність внести зміни) відповідно до процедури, описаної у внутрішніх інструкціях);

- результати повторних SW SCA / CR після усунення суперечностей (якщо виявлені під час SW SCA / CR)

Якщо виявляються невідповідності та / або помилки не виправляються, звіт повинен містити пояснення причин, за якими розробники коду ED залишались незмінними (виправленими) щодо кожного такого порушення або коментаря, підтверджене підписом розробника.

Звіт повинен бути представлений у доступній формі, зрозумілій для професіоналів, які не брали участі в SW SCA / CR.

## 1.4 Огляд основних процедур верифікації прикладної логіки спецплатформи RadiCS

Важливою частиною верифікації приладобудування та контролю (I&CS) є перевірка специфічної для системи логіки додатків(AL). Тут обговорюються завдання перевірки, характерні для AL для платформи RadICS на базі I&CS. Давайте коротко опишемо предмет перевірки.

AL як предмет перевірки.

Платформа RadICS складається з декількох типів модулів, заснованих на використанні мікросхем FPGA як обчислювальних, обробних та системних систем внутрішнього управління для кожного з модулів. З точки зору своєї функціональності та гнучкості на високому рівні, платформа RadICS є по суті захисним ПЛК, за винятком того, що внутрішня логіка виконується FPGA замість мікропроцесорів [4].

Функціональні можливості кожного модуля керуються загальною логікою платформи (PL), розробленою з використанням IDE, специфічної для FPGA, та реалізованою з мовами опису обладнання. Розробники платформи RadICS на базі I&CS не можуть впливати на PL, отже, перевірка PL не стосується їх. PL платформи RadICS перевірив постачальник платформи - RPC Radiy.

AL для I&CS розроблений за допомогою спеціалізованої IDE - інструмента конфігурації платформи Radiy (RPCT). RPCT забезпечує засоби для управління великими складними проєктами I&CS. AL реалізований з графічною мовою, специфічною для програми, яка дуже схожа на мову функціональної блок-схеми, зазначену в IEC 61131-3 [6]. На рисунку 1.2 нижче показаний фрагмент AL, розроблений у RPCT.

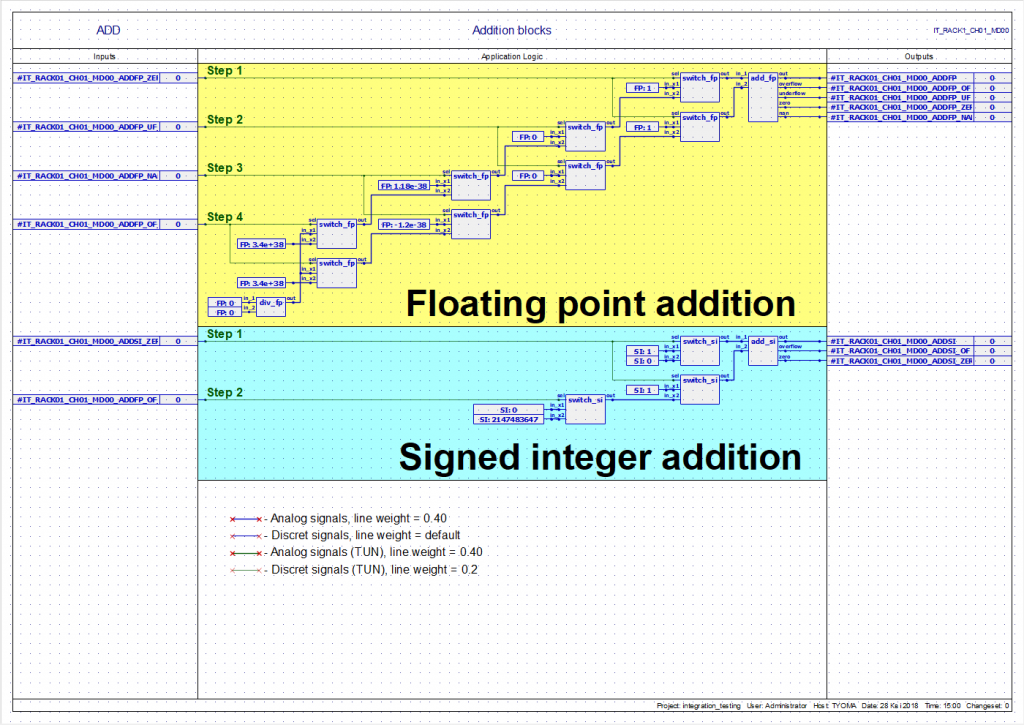


Рисунок 1.2 – Фрагмент логіки програми, розроблений у RPCT

Основними компонентами AL є функціональні блоки додатків (AFB). Кожен AFB дозволяє користувачеві вибирати та використовувати в рамках AL-проєктів певні функції: логічну, математичну, синхронізацію тощо. Кожна “графічна” AFB у RPCT прив’язується до апаратно реалізованого компонента AFB у логічному модулі (LM) на основі FPGA , цей компонент виконує зазначену функцію. Під час компіляції RPCT перетворює графічне представлення AL в код AL, який завантажується в LM і виконується під час роботи.

В основному код AL складається з наступних інструкцій програмного забезпечення:

1. Зчитування даних із виділених комірок пам'яті та передача їх у певний AFB.
2. Виконання функції AFB.
3. Запис результатів виконання функції AFB у виділені комірки пам'яті.

За структурою даних, AL Код - це шістнадцятковий код власного асемблера Radiy для LM на основі FPGA. Код AL також включає зручну для читання частину, яка не завантажується в LM і створена лише з інформативною метою. На рисунку 1.4.2 та 1.4.3 показані фрагменти обох частин.

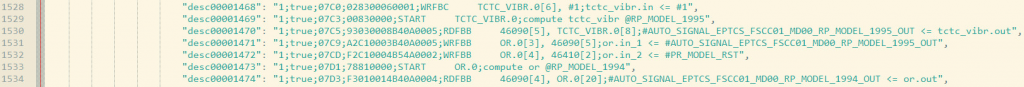


Рисунок 1.3 – Зрозуміла для людини частина

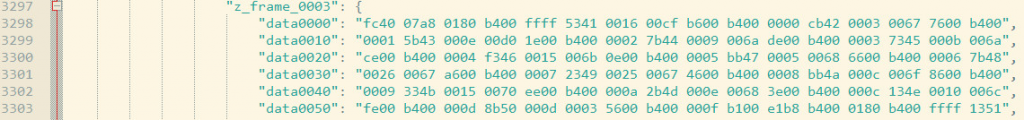


Рисунок 1.4 ­­­­­­­­ шістнадцятковий байт код

Рисунок 1.4.3 – Фрагменти логічного коду програми

Коди AL та AL є предметом процесу верифікації, який ми розглядаємо далі.

Нормативні вимоги, що впливають на перевірку AL

Нормативні документи NRC США, а також стандарти IEEE та IEC передбачають вимоги до діяльності, пов'язаної з програмним забезпеченням, та допоміжних процесів у життєвому циклі програмного забезпечення безпеки (SLC) комп'ютерних I&CS атомних електростанцій. Ці стандарти визначають різні типи програмного забезпечення, що стосується безпеки, та вимоги до SLC кожного типу програмного забезпечення. Структура SLC для розробки програмного забезпечення з використанням мов, орієнтованих на програми, є найбільш підходящою для проєктування та перевірки AL з RPCT. Наступний малюнок адаптований до рисунка 1.5 в IEC 60880 і ілюструє діяльність з верифікації та перевірки AL (V&V) [5].

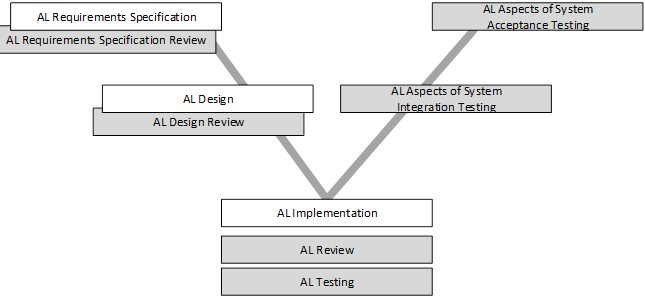


Рисунок 1.5 – Діяльність перевірки та перевірки логіки програми

Як вже було сказано раніше, конструкція AL не впливає на PL, тому для конкретного проєкту I&CS тільки AL потребує перевірки та перевірки.

Діяльність вищого рівня AL V&V, така як вимоги та верифікація проєкту, інтеграція системи та тестування прийняття, практично не передбачає сильної специфіки, визначеної платформою RadICS. Для того, щоб виконувати ці заходи найвищого рівня, організація, що розробляє, повинна дотримуватись відповідних вимог щодо V&V щодо комп'ютерних I&CS.

Перевірка впровадження АЛ тісно пов’язана із використовуваною мовою та середовищем дизайну. Тому ця діяльність вимагає використання конкретних інструментів та підходів.

Перевірка впровадження ЗВ може поєднувати різні завдання з перевірки, наприклад, аналіз, огляди, випробування або аудит (залежно від вимог регулюючого органу). Перегляд AL можна вважати найменшим обсягом зусиль з верифікації, як правило, цього може бути достатньо для додатків, що не стосуються безпеки. Поєднання огляду AL та тестування AL може розглядатися як достатня сфера для найбільш застосовних програм безпеки.

Огляд логіки програми.

Огляд AL надає об’єктивну оцінку Кодексу AL. Огляд AL визначає, чи повністю і правильно реалізує AL код належним чином проєкт AL, зазначений у проєктній документації. Огляд AL повинен проводитися як систематична перевірка програмного забезпечення.

З метою сприяння огляду AL, відділ верифікації RadICS використовує спеціальний інструмент - RPCT Outputs Verification Tool (ROVT).

Код AL складається з 2-х частин: двійкової частини, що завантажується до FSC LM, та текстової частини, яка є зручною для читання; обидва вони мають формат JSON (див. малюнок 2). Бінарну частину Кодексу AL неможливо прочитати людиною, тому не покладається на прямий людський огляд цієї частини AL. ROVT був незалежно розроблений відділом верифікації RadICS для перетворення двійкової частини коду AL у зручну для читання візуалізацію, що представляє задум проєкту, та порівняння обох частин для підтвердження їх еквівалентності.

ROVT надає такі ключові функції:

• ROVT може бути використаний як засіб пом'якшення, що дозволяє конструктору (або верифікатору) AL перевірити правильність компілятора (тобто, що бінарні файли правильно реалізують графічний логічний дизайн).

• ROVT надає можливість порівнювати проєкти AL та виділяти різницю у вхідних / вихідних сигналах та використанні AFB, ця функція може бути використана для оцінки змін під час планування тесту AL.

• ROVT здатний генерувати дерево шляху даних назад від обраного виходу до всіх входів, ця функція може бути використана для аналізу впливу під час планування тесту AL.

• ROVT виконує статичний аналіз AL. Перевірки статичного аналізу виконуються в автоматичному режимі відповідно до налаштованих користувачем правил безпеки.

Тестування логіки програми.

Метою тестування AL є забезпечення перевірки вимог до AL шляхом виконання програмних функціональних тестів. Тестування AL виконується за допомогою інструментального моделювання поведінки.

В даний час відділ верифікації RadICS використовує UAL Controller Test Framework (UCTF) для цілей тестування AL. UCTF містить шаблони для генерування тестових входів для тестів AL, тестовий стенд для виконання AL в імітованому середовищі та шаблони для визначення очікуваних результатів та оцінки очікуваних та фактичних результатів. UCTF реалізований у VHDL для інструменту моделювання ModelSim. Тому персонал групи верифікації повинен мати навички та знання з проєктування та моделювання VHDL.

Для подальшого використання Radiy розробляє спеціальний інструмент у наборі інструментів RPCT - Simulator. Симулятор виконує (тобто моделює) AL згідно із заданими користувачем сценаріями. Симулятор здатний імітувати визначені користувачем частини AL або всі AL.

Симулятор дозволить верифікатору визначати тестові сценарії безпосередньо в рамках RPCT, графічно представленого представлення AL. Simulator також підтримуватиме автоматизацію тестування мовами сценаріїв та надаватиме інтерфейси для інтеграції із зовнішніми системами (наприклад, системами моделювання заводу замовника).

# АНАЛІЗ ЗАСОБІВ АВТОМАТИЗАЦІЇ ВЕРИФІКАЦІЇ РЕЗУЛЬТАТІВ КОМПІЛЯЦІЇ ПЛАТФОРМИ RADICS

## 2.1 Програмні засоби верифікації результатів компіляції програмних проєктів критичних інформаційно-управляючих систем

Перевірка виходу програмних продуктів RadICS

Продукція RadICS використовується для розробки автоматизованих систем управління та захисту об'єктів атомної і теплової енергетики для ядерної безпеки. Програмне та апаратне забезпечення RadICS розроблено в рамках життєвого циклу безпеки (SLC) відповідно до найважливіших стандартів та правил безпеки (як IEEE, так і IEC). Компанія надає програмний продукт RPCT (Radiy Product Configuration Toolset), призначений для використання в ядерній промисловості. Він поєднує в собі багатокористувацький IDE, компілятор, який генерує вихідні файли для завантаження в апаратну платформу, системи налаштування та моніторингу для спостереження та обслуговування [7].

Файл потоку бітів (bitstream - , який створюється після успішної компіляції проєкту, призначений для завантаження на платформу I&C. Це робить компілятор RPCT інструментом, який має прямий вплив на систему безпеки, і результати роботи компілятора (у вигляді файлу спеціального формату) повинні бути перевірені перед завантаженням. Перевірка результатів компіляції проєктів RPCT (збірка) складається з трьох послідовних етапів (верифікаційних активностей).

Перш за все, верифікатор повинен переконатися, що процес компіляції був успішним, під час компіляції не виникало помилок, а всі файли у збірці не були пошкоджені після компіляції. Параметри проєкту та метрики повинні відповідати відповідним правилам керівництва з кодування RadICS. За результатами цієї активність формується документ - «Звіт про огляд результатів компіляції».

Якщо все в порядку, верифікатор виконує наступну активність - статичний аналіз коду (Static Code Analysis), що складається з послідовності додаткових перевірок у файлі, який потрібно завантажити на платформу I&C. Це включає перевірку правильності команд платформи, узгодженості файлу та інші дії.

Не менш важливої є активність - огляд програмного забезпечення (Code Review), що полягає у верифікації фактичних параметрів роботи програми з очікуваними значеннями. Успішні результати виконання всіх трьох активностей (без виявлених аномалій) дозволяє завантажувати перевірену збірку на платформу I&C (і перед завантаженням слід дотримуватися багато додаткових процедур).

SCA/CR під час процесу верифікації можна виконати вручну (без будь-яких спеціальних інструментів), але це непросте завдання: хоча файл потоку бітів читається (він представлений у форматі JSON), його структуру та вміст досить важко прочитати та зрозуміти. Програмне забезпечення ROVT (RPCT Outputs Verification Tool) було розроблено як додатковий інструмент, що має підвищити якість та ефективність вищезазначених процесів верифікації.

Метою використання ROVT є швидкий аналіз проєкту та багато автоматичних перевірок статичного аналізу. ROVT виконує аналіз результатів компіляції проєктів (збірки) для створення списків об'єктів збірки (сигнали, функціональні блоки, елементи обладнання тощо). Також аналізуються властивості кожного об'єкта, а також відносини між ними. В результаті зворотного проєктування збірки відновлюється конструкція, яка дорівнює початковому проєкту. Створена структура відображається у вигляді набору звітів, які може переглянути особа, яка виконує перевірку збірки. Ці звіти також можуть бути використані як частина документів з результатів верифікації.



Рисунок 2.1 - Інтерфейс початкової сторінки ROVT

ROVT забезпечує кілька потоків даних залежно від намірів користувача. У наступному списку описані найбільш часто використовувані режими роботи ROVT (наведені з найпростіших):

Генерація основних звітів (у вигляді набору файлів, без перевірок статичного аналізу коду (SСA));

Основні звіти з генерацією перевірок SСA (як набір файлів);

Формування та огляд звітів (усередині ROVT, генерація рецензованих звітів у вигляді набору файлів);

Злиття звітів (для об’єднання результатів огляду кількома верифікаторами);

Порівняння збірок проєкту (створення дельта-звіту зі списком розбіжностей між різними збірками одного і того ж проєкту).

Будь-який тип генерації звітів можна зберегти у файлі проєкту ROVT на будь-якому етапі перевірки, щоб продовжити роботу пізніше або поділитися результатами з колегами.

Звіти, сформовані ROVT, включають:

Звіт про результати компіляції - містить аналіз журналу компіляції, результати перевірки цілісності збірки, споживання ресурсів апаратного забезпечення платформи тощо.

Логічний звіт програми користувача - містить елементи, що використовуються для визначення алгоритмів: вхідні та вихідні сигнали, функціональні блоки, оптичні з'єднання тощо.

Апаратний звіт - містить конфігурацію апаратних модулів, вхідні та вихідні порти, налаштування мережі тощо.

Звіт про налаштування - містить інформацію про задані значення алгоритму, сигнали, значення яких можна змінювати в режимі настройки платформи.

Дельта-звіт - містить перелік різних функцій між двома збірками проєктів: логічні та апаратні елементи або їх властивості (створюється лише в режимі «Порівняння» ROVT).

Звіт про залежності - містить перелік вхідних логічних сигналів програми, які впливають на конкретний вихідний сигнал. Цей звіт корисний для аналізу впливу та оцінки тестового покриття (створюється лише у тому випадку, якщо був увімкнений офіційний режим охоплення ROVT).

Реалізація процесів SCA в ROVT.

Перевірка результатів компіляції RPCT в основному означає виявлення можливих помилок компілятора RPCT (або конструктора RPCT). Для виявлення таких помилок використовується кілька типів перевірок:

Порівняння фрагментів даних збірки (для виявлення помилок узгодженості збірки);

Порівняння даних побудови та заздалегідь визначених обмежень (як динамічно змінюваних параметрів статичного аналізу, так і заздалегідь визначених параметрів платформи);

Порівняння даних збірки та проєктної документації (ця виконується верифікатором вручну).

Наступна частина опису деяких з цих перевірок.

Очевидно, що немає жодних гарантій того, що дані збірки є безпечними, якщо збірка була змінена після компіляції. ROVT обчислює MD5 і розмір кожного файлу збірки і порівнює їх зі значеннями, наведеними в даних збірки. У разі будь-якої невідповідності, збірка проєкту вважається пошкодженою і не може бути використана для будь-яких програм безпеки.

Описана перевірка може запобігти використанню збірки, яка була пошкоджена після компіляції, але що, якщо компілятор сам генерує неправильний код? Файл збірки для завантаження на платформу складається з двох рівноцінних частин - складного, але читабельного тексту та нечитабельних шістнадцяткових фреймів даних. Деякі логічні помилки компілятора можуть бути виявлені при порівнянні цих частин. ROVT використовує текстову частину для створення власної версії шістнадцяткової частини, а потім робить порівняння фактичного та очікуваного вмісту. У разі будь-якої невідповідності узгодженість збірки проєкту вважається непідтвердженою, і дані збірки не можуть використовуватися для будь-яких програм безпеки.

Конструктор RPCT може розробити гнучку та широку логіку додатків, використовуючи набір попередньо розроблених елементів - Функціональні блоки додатків (AFB). Ці блоки також гнучкі, деякі з них мають регульовані параметри, а більшість із них мають додаткові порти звітування про винятки. RPCT забезпечує придушення таких портів для логічного конструктора (це все ще не рекомендується для проєктів, що стосуються безпеки). Проблема полягає в тому, як забезпечити, щоб: а) виключені лише віртуальні порти; б) кожен підключаємий порт був включений до даних збірки. ROVT містить внутрішньо незалежно розроблену базу даних можливих конфігурацій AFB, і кожен елемент AFB у файлі бітового потоку порівнюється з обмеженнями бази даних. Якщо порт AFB не включений у бітовий потік, але цей порт є необов’язковим, генерується відповідне попередження, тому верифікатор повинен переконатися, що цей порт не вимагається конструкцією. Якщо порт AFB не включений у бітовий потік, і цей порт не є обов'язковим, генерується помилка, оскільки компілятор RPCT має неправильну модель AFB.

Логіка додатків у RPCT розроблена як поєднання функціональних блоків та з'єднань. Більшість функціональних елементів призначені для обробки певного типу даних: Беззнакове ціле число, плаваюча точка або дискретне, тому вони мають порти вводу / виводу певного типу даних. Різниця між ними суттєва, тому зв’язок між портами різного типу є неправильним, і RPCT не дозволить успішно скомпілювати таке з’єднання. Щоб переконатися, що компілятор не генерує несумісні з'єднання, ROVT перевіряє всі з'єднання між функціональними елементами та додає відповідне повідомлення про помилку, якщо порти одного і того ж з'єднання мають різний тип даних.

Цифрова інформаційно-управляюча платформа Radics дозволяє користувачеві налаштовувати заздалегідь визначені точки входу алгоритму в заздалегідь визначені межі, коли система не працює (логічний модуль переведено в режим налаштування). Такі точки називаються сигналами налаштування (їх також можна назвати заданими значеннями) і мають три важливі значення: Висока межа, Низька межа та Значення за замовчуванням. ROVT перевіряє, чи ці значення сумісні (висока межа більша, ніж нижня, а значення за замовчуванням знаходиться між межами).

Інша корисна перевірка - це аналіз потенційної петлі. Циклічне відтворення виникає, коли функціональний елемент A вимагає виведення функціонального елемента B для обчислення результату A, але функціональний елемент B вимагає вихід A для обчислення результату B (рис. 2.2).

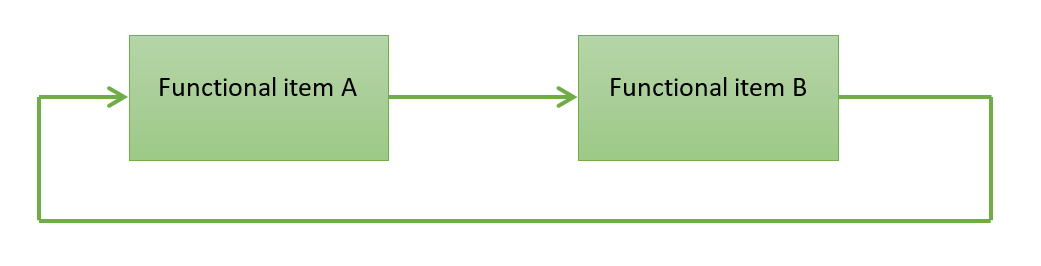


Рисунок 2.2 – Приклад схематичного зворотного зв'язку

Така поведінка може плануватися в деяких алгоритмах, але будь-який зворотний зв'язок вимагає пильної уваги та ретельного огляду. ROVT виявляє описані ситуації та генерує попереджувальні повідомлення для перевірки.

Остання частина функцій статичного аналізу ROVT - регульовані параметри, які дозволяють верифікатору переконатися, що дані збірки відповідають обмеженням керівництва кодування. ROVT може перевірити, чи довжина імені елементів проєкту не перевищує визначених меж, або ім'я містить заборонені символи. ROVT також може оцінювати складність обчислення: верифікатор може вказати максимальну довжину одного ланцюжка обчислення (набір підключених функціональних елементів від вхідного сигналу до вихідного сигналу) та інші метрики складності.

Перевірка статичним аналізом не може здійснити повну верифікацію результатів, однак багато дій людини значно спрощуються завдяки ROVT.

## 2.2 Визначення можливих напрямків удосконалення функціоналу ROVT.

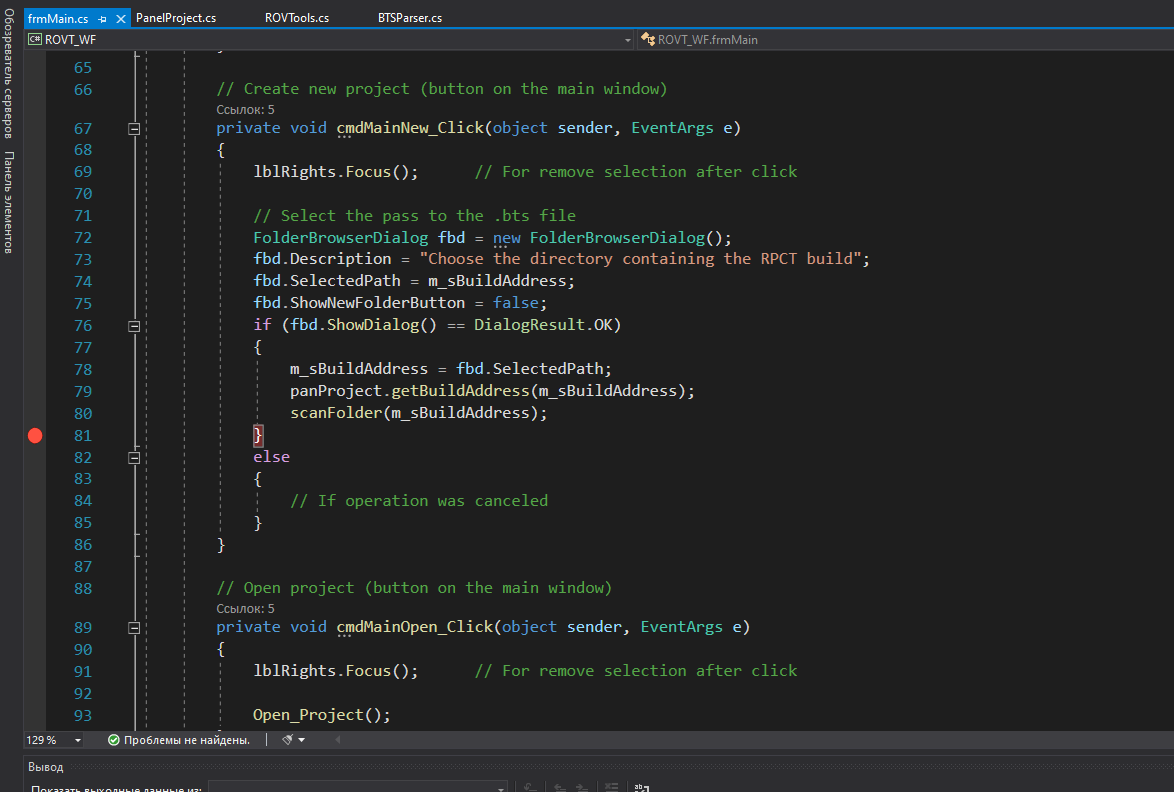
Під час запуску ROVT виникає тимчасова затримка. На деяких проєктах вона доходить до півгодини. Були проаналізовані різні частини програми і з'ясувалося в якій саме частині проєкту витрачається найбільше часу. Для цього розділили ROVT на 3 основні частини: підготовка (пошук і перевірка на їх коректність) файлів для роботи, парсинг даних і фінальне - побудова звітів (для докладної інформації дивитися пункт 2.1) Для цього поставили контрольні точки з таймером в місцях, де можливе уповільнення програми. Перший breakpoint поставили на 81 рядок у класі frmMain, після методу scanFolder, який виконує сканування всіх файлів в папці і перевіряє їх на коректність (рис. 2.3).

Рисунок 2.3 – Перший breakpoint

Другий breakpoint поставили на 97 рядку в класі ROVTools, після методу fullParse, який парсить дані в файлах просканованих раніше (рис. 2.4).

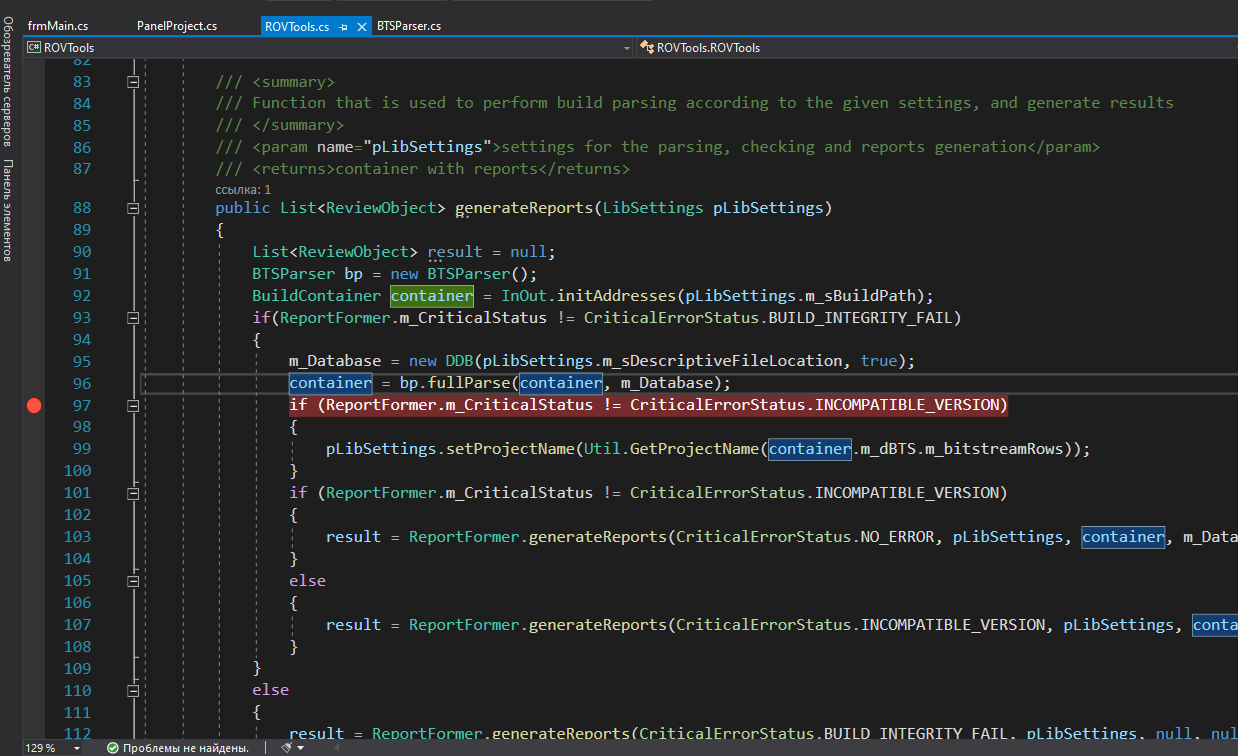


Рисунок 2.4 – Другий breakpoint

Третій breakpoint поставили на 136 рядку в класі PanelProject, після методу generateReport, який відповідає за побудову звітів.

Після виконання програми на невеликому проєкті, з'ясувалося, що основне навантаження припадає саме на парсинг даних.

## 2.3 Пропозиції щодо підвищення ефективності функціоналу ROVT.

Розглянемо кілька варіантів збільшення продуктивності програми.

Оновлення ПЗ.

Перехід на останню версію якої-небудь використовуваної в проєкті бібліотеки, СУБД, віртуальної машини Erlang'а або ядра Linux може істотно збільшити швидкість роботи програми. Просте і, як правило, швидке рішення.

У нашій програмі використовується кілька сторонніх бібліотек, при перевірці яких на наявність оновлення, з'ясувалося, що всі вони останньої версії.

Налаштування оточення.

Використовувана СУБД або операційна система можуть бути налаштовані неправильно. Налаштування за замовчуванням MySQL і PostgreSQL припускають, що ви намагаєтеся запустити СУБД на першому гібридному процесорі x86. Цей метод навіть простіше, ніж оновлення ПЗ. Однак застосовувати його, зі зрозумілих причин, потрібно після оновлення. Або в разі, якщо оновлення з якихось причин в доступному для огляду майбутньому неможливо. У нас в проєкті не використовуються бази даних.

Видалення непотрібного функціонала.

Можна збільшити швидкодію додатку, викинувши непотрібний код. Іноді виявляється, що програма робить щось непотрібне або не дуже потрібне. Можливо, одна з проблем, що вирішуються втратила свою актуальність. Іноді замовник замість справжньої проблеми описує програмісту своє бачення її вирішення, а програміст в силу своєї недосвідченості просто кодує це рішення. Тим часом рішення справжньої проблеми може бути набагато простіше. Іноді якийсь функціонал обростає милицями і підпорами. У цьому випадку має сенс реалізувати функціонал з чистого аркуша, а старе рішення викинути. Після аналізу коду, з'ясувалося, що весь функціонал є актуальним і необхідним.

Купівля нового заліза.

Часто набагато швидше і дешевше купити нове залізо, ніж оптимізувати код програми. У ряді випадків подвоєння числа ядер процесора може привести до подвоєння швидкості роботи програми. Можна докупити оперативної пам'яті і зберігати дані в ній, замість того, щоб брати їх з диска або передавати по мережі. Можна перенести базу даних на SSD. Якщо програма масштабується горизонтально, можна докупити десяток серверів. Так як бюджет відділу дуже обмежений, цей варіант не підходить.

Мемоізація і кешування.

Мемоізація - це прийом збереження проміжних результатів, які можуть ще раз знадобитися найближчим часом, щоб уникнути їх повторного обчислення. Цей прийом можна розглядати як різновид кешування. Кешування- це збереження результатів чого завгодно. Наприклад, кешуватися можуть веб-сторінки або звіти за місяць. Кешування може не застосовуватися, якщо кешировані дані швидко оновлюються. Також в контексті кешування часто постає проблема інвалідаціі кеша. В контексті мемоізаціі така проблема не виникає, так як мемоізаціі зазвичай піддаються чисті функції, тобто, функції без побічних ефектів, значення яких залежить тільки від аргументів. Мемоізація і кешування ефективні і легко реалізуються, проте неправильне кешування може перешкоджати горизонтальному масштабуванню програми. При додаванні в додаток чергового кеша, потрібно думати про те, як їм управляти, коли програма буде працювати в двох і більше екземплярах.

Даний метод не застосуємо до поточної реалізації, так як використовуються виключно брудні функції і запам'ятовування минулого результату не потрібно.

Розпаралелювання, багатопоточність .

Розпаралелювання може бути простою або складною операцією, в залежності від складності логіки всередині. Наприклад, в Erlang дуже багато завдань можуть бути з легкістю розпараленені шляхом написання буквально десятка рядків коду. А в Scala ви можете без особливих зусиль скористатися паралельними колекціями замість звичайних. Однак деякі завдання не можуть бути вирішені паралельно за своєю природою. І якщо програма працює на одноядерному процесорі, розпаралелювання нічого не дасть. Недетерміновані функції і функції з побічними ефектами ускладнюють застосування цієї оптимізації. При написанні web-додатків розпаралелювання не завжди можливо, тому що не можна зайняти всі ядра обробкою запиту одного користувача, заблокувавши тим самим обробку інших запитів.

Повністю підходить для нашого випадку. Так як в даний момент програма працює виключно в одному потоці, потрібно задіяти інші потужності процесора.

Розподіл навантаження.

Якщо навантаження на СУБД мала, можна скористатися тригерами, розвантаживши тим самим сам додаток і зменшивши трафік. Або, навпаки, можна перенести всю логіку в додаток, розвантаживши СУБД. Для побудови звітів, створення резервних копій і виконання інших важких операцій над СУБД має сенс завести спеціальну репліку. СУБД можна налаштувати так, щоб різні таблиці зберігалися на різних фізичних дисках. Можна віддати користувачеві статичну сторінку з JavaScript і спілкуватися з ним виключно за допомогою REST API. Нехай сам генерує собі HTML. Статичний контент можна тримати на окремому домені. Цим ви зменшите трафік, так як на цей домен не будуть відправлятися кук-файли. Нема чого gzip'овать / шифрувати дані в Apache або навіть в самому додатку, якщо з цим завданням набагато краще впорається nginx. За допомогою шардінга можна розподілити навантаження між декількома репліками бази даних, процесами Erlang'а або екземплярами Memcached .

В програмі не використовується база даних.

Ледачі обчислення.

Ледачі обчислення - це коли замість конкретного значення повертається анонімна функція, яка при виклику обчислює це значення. У ряді мов програмування ледачі обчислення підтримуються на рівні синтаксису. Фокус в тому, щоб значення було обчислено безпосередньо перед його використанням. Уявіть собі ситуацію, коли ми віддаємо дані в форматі CSV і користувач може задати фільтр, який визначає, які стовпчики мають бути передані. В цьому випадку ледачі обчислення виявляються дуже доречними. Якщо виявиться, що значення насправді не потрібно, ми заощадимо час, який було б витрачено на його обчислення. Однак слід зазначити, що ледачі обчислення призводять до збільшення обсягу використовуваної пам'яті і можуть погано працювати з брудними функціями.

Як з'ясувалося в пункті 2.2 проблема з продуктивністю виникає при парсінгу bitstream, а для цього завдання не можна використовувати ледачі обчислення.

Відкладені розрахунки. Навіщо обчислювати щось прямо зараз, якщо це можна зробити потім? При обробці HTTP-запиту ми можемо моментально повернути користувачеві OK, а безпосередню роботу виконати в фоновому процесі. Якщо запит дуже важливий, ми можемо покласти його в персистентную чергу завдань, оброблювану за cron'у або групою безперервно працюючіх процесів. В останньому випадку ми навіть маємо хороші шанси отримати горизонтальне масштабування і, відповідно, реальне збільшення швидкості, а не тільки видиме. Крім того, відкладені завдання можуть бути схожі. Наприклад, їм потрібні одні і ті ж дані з БД. В цьому випадку при відкладеній обробці N завдань однієї пачкою можна сходити в базу в N раз менше.

Цей пункт не підходить, так як специфіка роботи парсинга передбачає отримання результату негайно.

Більш відповідні алгоритми і структури даних.

Quicksort швидше, ніж сортування бульбашкою, а еліптичні криві швидше RSA. Якщо потрібно перевірити приналежність елемента множині, слід використовувати хеш-таблиці, а не однозв'язні списки. Правильні індекси і денормалізация схеми бази даних можуть істотно скоротити час виконання SQL-запитів. Якщо потрібно синхронізувати деякі дані, замість повного їх пересилання при кожній зміні краще використовувати схему снапшот + апдейти.

Метод не підходить, так при парсингу використовуються атомарні і досить прості алгоритми.

Апроксимація.

Апроксимація - це метод свідомого спрощення надмірно складного теоретичного знання з метою привести його у відповідність до потреб і можливостей практики. Це майже те саме, що й випадок більш відповідного алгоритму, тільки з втратою точності. Замість довгої арифметики часто можна обійтися звичайними float'ами. При зборі статистики дані можна слати по UDP замість TCP. Нехай невелика частина пакетів не дійде, а частина - прийде двічі. При зборі статистики набагато важливіша зміна цифр, а не конкретні значення. Також, наприклад, нема чого будувати графік по всіх точках, якщо можна взяти їх підмножину і побудувати криву Безьє. Замість дорогого обчислення медіани часто можна порахувати середнє.

Даний пункт неможливо реалізувати, через і так просту логіку виконання парсинга .

Переписування на іншу мову.

Цілком може виявитися, що програму в істотному ступені гальмує прибирання сміття або, скажімо, перевірка типів на етапі виконання. Переписування невеликих частин програми з Ruby на Scala або з Erlang на OCaml може привести до прискорення цієї програми. Якщо переписується шматок коду досить простий, можна з невеликим ризиком переписати його на С або C ++. Цей метод потрібно використовувати вкрай обережно. Він призводить до появи зоопарку мов програмування, що ускладнює підтримку проєкту. Метод може не спрацювати, наприклад, через додаткові витрати на перетворення даних з одного подання до іншого. Також він може бути небезпечним. Наприклад, помилка в NIF може привести до падіння всієї віртуальної машини Erlang'а, а не одного процесу.

Даний варіант неможливий, так як переписання частин коду, в нашому випадку парсинг потребує великих витрат часу з боку команди і непрогнозований результат. І цей пункт прямо суперечить стандартам написання коду в компанії.

# 3 ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ROVT ЗА РАХУНОК РЕАЛІЗАЦІЇ БАГАТОПОТОЧНОСТІ.

## 3.1 Аналіз способів реалізації багатопоточної роботи прикладного програмного забезпечення.

Як показано в розділі 2.3 для вирішення проблеми продуктивності потрібно використовувати багатопоточність. Далі розглянемо цей метод і способи його реалізації детальніше.

Багатопоточність - здатність центрального процесора або одного ядра в багатоядерному процесорі одночасно виконувати кілька потоків, відповідним чином підтримуючих операційною системою.

В сучасних операційних системах одночасно працює безліч процесів. Навантаження від операційної системи на процесор йде по так званому конвеєру, на який «накладаються» потрібні завдання для ядра. До ядра по конвеєру надходять завдання, що вимагають виконавчої потужності, на які витрачається процесорний час.

У тих випадках, коли багатопроцесорні системи включають в себе кілька повних блоків обробки, багатопоточность спрямована на максимальне використання ресурсів одного ядра, використовуючи паралелізм на рівні потоків, а також на рівні інструкцій. Оскільки ці два методи є взаємодоповнюючими, їх іноді об'єднують в системах з декількома багатопотоковими ЦП і в ЦП з декількома багатопотоковими ядрами.

Часто трапляються випадки, коли для виконання необхідної операції процесору доводиться чекати дані з кеша на більш низькій швидкості (L3 кеш ), або ж оперативної пам'яті. Дана ситуація називається кеш-промах. Це відбувається, коли в кеші ядра була знайдена така інформація і доводиться звертатися до більш повільної пам'яті. Також існують і інші причини, що змушують переривати виконання операції ядром, що негативно позначається на продуктивності.

Сфери застосування багатопоточних процесорів.

З розвитком комп'ютерних технологій перелік програм, що використовують багатопоточність, неухильно зростає. Це дає величезний простір розробникам для створення нового софта та ігор. Наприклад, зараз кожен сучасний triple -A проєкт оптимізований для багатопотокових процесорів, що дозволяє насолоджуватися грою, отримуючи високий рівень fps на багатоядерному CPU.

Ще більше поширені багатоядерні системи в середовищі розробників. Програми для 3D-моделювання, монтажу відео і створення музики вимагають паралельного виконання великої кількості завдань, з чим добре справляються системи з Hyper-Threading або SMT. В операційних системах потужність одного потоку може витрачатися на фонові завдання ( Skype, браузер, месенджер), в той час як інші задіюються для важкої гри або програми.

Але далеко не завжди збільшення кількості потоків означає збільшення загальної продуктивності. Чому ж SMT процесори часом поступаються небагатопоточним побратимам? Справа в програмній підтримці. Іноді погано оптимізовані програми не можуть відрізняти логічний потік від справжнього ядра, через що на одне ядро може потрапити два важких завдання і уповільнити його роботу. Проте, подібні технології мають величезний потенціал, головне - грамотно реалізувати його на програмному рівні.

Існує кілька причин використання багатопоточності. Наприклад, програма здійснює запит на деякий сервер мережі, що може зайняти якийсь час. Навряд чи ви хочете, щоб користувальницький інтерфейс був заблокований, і користувачеві просто довелося зачекати хвилину, поки відповідь сервера повернеться. На даний момент користувач може виконати деякі інші дії або навіть скасувати запит, надісланий на сервер. У цих ситуаціях використання багатопотоковості є вигідним.

Для всіх видів діяльності, які вимагають очікування, наприклад, через отримання доступу до файлу, бази даних або мережі, може запускатися новий потік, що дозволяє виконувати інші завдання одночасно. Багатопотоковість може допомогти, навіть якщо задіяні лише обробні завдання. Кілька потоків одного і того ж процесу можуть виконуватися одночасно різними центральними процесорами або, що частіше в наші дні, різними ядрами одного і того ж багатоядерного процесора.

Необхідно знати особливості одночасного виконання багатьох потоків. Оскільки вони працюють одночасно, проблеми можуть виникнути, коли вони отримують доступ до одних і тих самих даних. Щоб цього не сталося, необхідно впровадити механізми синхронізації.

Основи багатопотокової обробки в середовищі. Мережа.

Існує два типи багатозадачності: на основі процесів і потоків. У зв’язку з цим важливо розуміти відмінності між ними. Процес відповідає за управління ресурсами, такими як віртуальна пам’ять та дескриптори Windows, і містить принаймні один потік. Наявність принаймні одного потоку необхідна для виконання будь-якої програми. Тому багатозадачність на основі процесів - це засіб, за допомогою якого дві або більше програм можуть одночасно працювати на комп’ютері.

Наприклад, багатозадачність на основі процесів дозволяє одночасно запускати програми обробки текстів, електронних таблиць та перегляду веб-сторінок. У багатозадачності на основі процесів програма є найменшою одиницею коду, яку може координувати планувальник завдань.

Потік - це скоординована одиниця виконуваного коду. Цей термін своїм походженням зобов'язаний поняттю "потік виконання". За допомогою багатозадачності на основі потоків кожен процес повинен мати принаймні один потік, хоча їх може бути і більше. Це означає, що два або більше завдань можна одночасно вирішувати в одній програмі. Наприклад, текст можна відформатувати в текстовому редакторі одночасно з його друком, за умови, що обидві ці дії виконуються у двох окремих потоках.

Різниці в багатозадачності на основі процесів і потоків можна узагальнити наступним чином: багатозадачність на основі процесів організована для паралельного виконання програм, тоді як багатозадачність на основі потоків організована для паралельного виконання окремих частин програми.

Головною перевагою багатопотоковості є те, що вона дозволяє писати програми, які працюють дуже ефективно, використовуючи переваги простою, який неминуче виникає під час виконання більшості програм. Як відомо, більшість пристроїв вводу-виводу, будь то пристрої, підключені до мережевих портів, дисководів або клавіатур, набагато повільніші, ніж центральний процесор (ЦП). Тому більшу частину свого часу програмі доводиться чекати, поки дані будуть надіслані на пристрій вводу-виводу або отримані з нього. А завдяки багатопотоковій обробці програма може вирішити деякі інші проблеми під час вимушеного простою.

Наприклад, поки одна частина програми надсилає файл через Інтернет, інша частина може читати текстову інформацію, введену з клавіатури, а третя частина може буферизувати наступний блок даних, що надсилаються.

Потік може бути в одному з декількох станів. Загалом, потік може бути виконуваним; готовий до виконання, як тільки отримає час і ресурси Центрального банку; припинено, тобто тимчасово виконується; відновлений пізніше; заблоковано очікування виконання ресурсів; а також припинення, коли його виконання закінчено і не може бути відновлене.

## 3.2 Реалізація багатопоточності на мові програмування C#.

Приклади використання в навколишньому середовищі .Net.

Потік - це незалежна послідовність інструкцій у програмі. Потоки необхідні як для клієнтських, так і для серверних додатків. Наприклад, коли ви вводите якийсь код C# у вікні редактора Visual Studio, він аналізує різні синтаксичні помилки. Цей аналіз виконується окремим фоновим потоком. Те саме відбувається в засобі перевірки правопису в Microsoft Word. Один потік чекає введення користувачем, тоді як інший робить аналіз у фоновому режимі. Третій потік може зберігати записані дані у тимчасовому файлі, а четвертий - завантажувати додаткові дані з Інтернету.

У програмі, яка працює на сервері, один потік завжди чекає запиту від клієнта і тому називається потоком прослуховування. Отримавши запит, він негайно надсилає його в окремий робочий потік, який потім сам продовжує взаємодіяти з клієнтом. Після цього потоку прослуховувача потрібно відшкодувати свої обов'язки в очікуванні наступного запиту від наступного клієнта.

Кожен процес складається з таких ресурсів, як дескриптори вікон, дескриптори файлів та інші об'єкти ядра, має виділену область у віртуальній пам'яті та містить принаймні один потік. Потоки планується виконувати операційною системою. Будь-який потік має пріоритет, лічильник інструкцій, який вказує на місце в програмі, де відбувається обробка, і стек, який зберігає локальні змінні потоку. Стек кожного потоку виглядає по-різному, але пам'ять програмного коду та купа спільно використовуються між усіма потоками, що працюють в рамках одного процесу.

Це дозволяє потокам в рамках одного процесу швидко взаємодіяти між собою, оскільки всі потоки в процесі отримують доступ до однієї і тієї ж віртуальної пам'яті. Однак це також ускладнює справи, оскільки дозволяє декільком потокам змінювати одну і ту ж область пам'яті.

Пул потоків CLR.

На створення потоку потрібен час. Якщо потрібно виконати різні короткі завдання, ви можете заздалегідь створити набір потоків, а потім просто надіслати відповідні запити, коли настане черга їх виконувати. Було б непогано, якби число цих потоків автоматично зростало, оскільки попит на потоки збільшувався і зменшувався, коли виникає потреба у звільненні ресурсів.

Вам не потрібно самостійно створювати такий список потоків. Для управління таким списком надається клас ThreadPool, який за необхідності зменшує та збільшує кількість потоків у пулі до максимально дозволеної. Значення максимально дозволеної кількості потоків у пулі може бути змінено [10]. У випадку двоядерного процесора за замовчуванням він має 1023 робочі потоки та 1000 потоків вводу-виводу [8].

Ви можете вказати мінімальну кількість потоків, які починаються відразу після створення пулу, і максимальну кількість потоків, доступних у пулі. Якщо залишилось трохи стати суб’єктом обробки завдання, і максимальна кількість потоків у пулі вже досягнуто, тоді нові завдання будуть поміщені в чергу і чекатимуть там, поки один із потоків завершить свою роботу.

Ви можете використовувати метод QueueUserWorkItem (), щоб запросити об’єднаний потік для обробки виклику методу. Цей метод перевантажений, щоб дозволити необов’язковий параметр System.Object для даних спеціального стану на додаток до екземпляра делегата WaitCallback.

Нижче наведено зразок програми, яка спочатку зчитує та друкує інформацію про максимальну кількість робочих потоків та потоків вводу-виводу. Потім у циклі for JobForAThread () присвоюється потоку з пулу потоків, викликаючи ThreadPool.QueueUserWorkltem () та передаючи делегат WaitCallback. Пул потоків отримує цей запит і вибирає один із потоків із пулу для виклику методу. Якщо пул ще не існує, він створюється, і запускається перший потік. Якщо пул існує і в ньому є один вільний потік, завдання перенаправляється на цей потік.

Впровадження багатопоточності в C#.

.NET Framework визначає два варіанти потоків: передній та задній. За замовчуванням створений потік автоматично виключається, але ви можете зробити його фоновим. Єдина відмінність між потоками переднього та фонового режимів полягає в тому, що фоновий потік автоматично припиняється, якщо всі потоки переднього плану зупиняються в процесі.

У зв'язку з організацією багатозадачності на основі потоків існує необхідність у спеціальному режимі, який називається синхронізацією, і дозволяє дуже конкретно координувати виконання потоків. Для такої синхронізації C# забезпечує окрему підсистему.

Всі процеси складаються щонайменше з одного потоку, який зазвичай називають основним потоком, оскільки саме з цього потоку починається виконання програми. Інші потоки можна створити з основного потоку.

Багатозадачність як на основі процесів, так і на основі потоків підтримується в C# та .NET Framework. Отже, використовуючи C#, ви можете створювати як процеси, так і потоки, а також керувати обома. Для того, щоб розпочати новий процес, потрібно дуже мало зусиль від програми, оскільки кожен попередній процес повністю відокремлений від наступного.

Набагато важливішою є підтримка багатопоточності на C#, що полегшує написання високопродуктивних багатопоточних програм на C# порівняно з деякими іншими мовами програмування.

Класи, що підтримують багатопоточність, визначені у просторі імен System.Threading.

Простір імен System.Threading містить різні типи, що дозволяють створювати багатопотокові програми. Мабуть, найважливішим серед них є клас Thread, оскільки він представляє окрему нитку. Щоб програмно отримати посилання на потік, який виконується певним його екземпляром, потрібно викликати статичну властивість Thread.CurrentThread.

Окремий потік також можна в будь-який час перемістити в певний контекст, і він може переміщатися в новому контексті за забаганням CLR. Щоб отримати поточний контекст, у якому запущений потік, використовуйте властивість static Thread. CurrentContext (який повертає об'єкт System.Runtime.Remoting.Contexts.Context):

## 3.3 Імплементація вибраних засобів багатопоточності в ROVT.

В результаті проведеного аналізу (пункт 2.3) з'ясувалося, що основна затримка при обробці проєкту виникає на етапі парсинга bitstream. Значить впровадити багатопоточність потрібно саме там. Перед початком імплементації потрібно розібратися як працює поточна реалізація парсинга і з яких основних частин вона складається.

З пункту 2.1 парсинг ділитися на основні 3 частини: обробка UAL (User Application Logic), обробка HW (HardWare), обробка TUN (Tunning). З пункту 1.4 необхідна інформація для роботи програми в bitstream складається з subsystem (підсистем), а вони в свою чергу складаються з каналів, тобто на виході ми отримуємо імпровізоване дерево. Так як прямих кордонів на його розширень нема (ось цю фразу треба точно переосмислити) і чим більше підсистем, тим довше доведеться парсити bitstream. Як описано в пункті 3.1 для прискорення процесу парсинга потрібно використовувати вільні потоки з thread pool. За структурою методу з'ясувалося, що більш правильно буде покрити «дерево» bitstream деревом потоків, яке буде динамічно розширюватись, при зростанні кількості підсистем і каналів в проєкті РПСТ. Підсумуємо: на кожну підсистему виділяємо потік, в кожному з цих потоків виділяємо по потоку на кожен вид звіту (UAL, HW, TUN), в кожному з цих потоків виділяємо по потоку на кожен канал.

Алгоритм реалізації багатопоточності в ROVT.

У методі fullParse створюється масив екземплярів класу Task (пункт 3.1), цей клас виділяє потік з пулу потоків для виконання методу або дії [9].

При оголошенні масиву, вказується його розмірність, а це кількість підсистем в проєкті РПСТ (рис. 3.1).

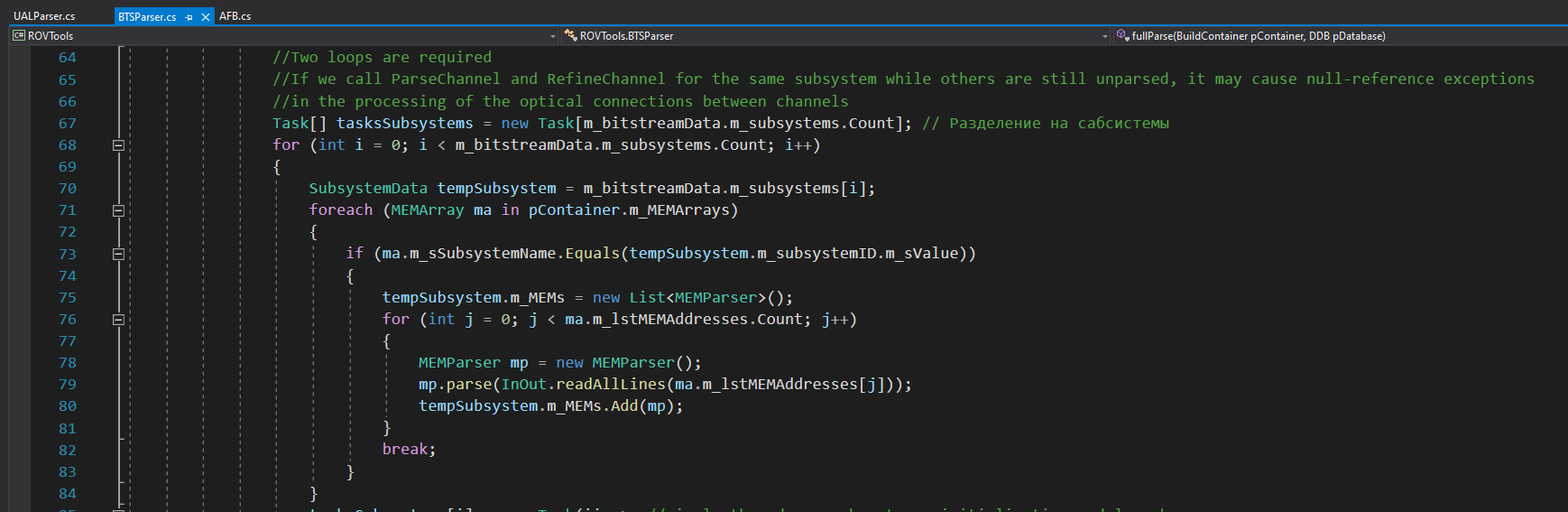


Рисунок 3.1

Потім в циклі, який проходиться по всіх підсистемам інициалізуєм і запускаємо кожен потік. При ініціалізації передаємо в конструктор класу Task лямбда вираз, в якому вказується які дії потрібно виконати потоку і другому параметром передаємо ітератор циклу (рис. 3.2).

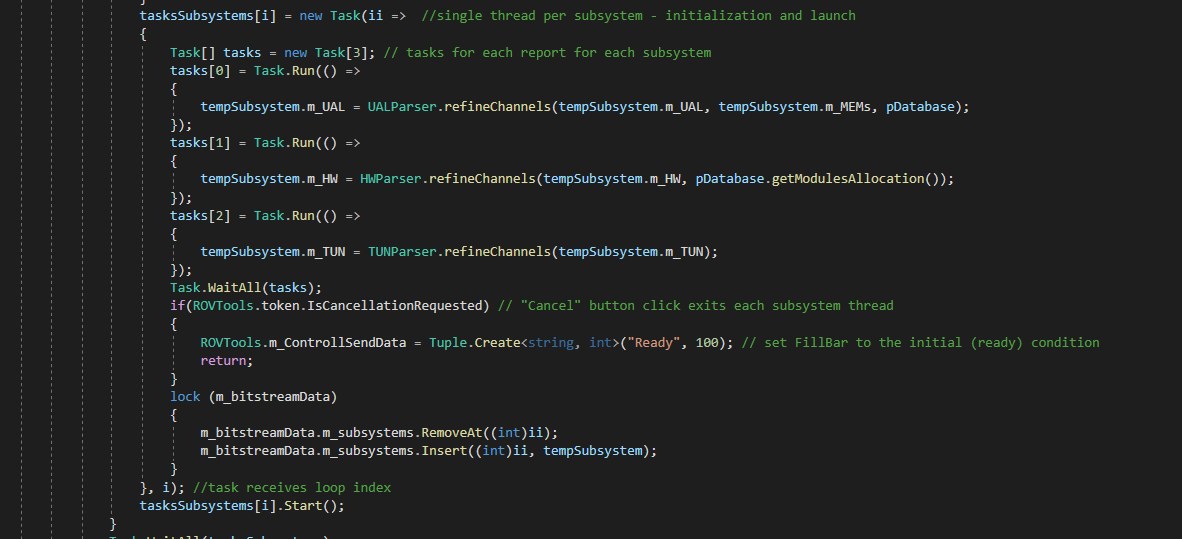


Рисунок 3.2

Таким чином, здійснюється розподіл потоків на першому рівні дерева, тобто в області підсистем.

В лямбда виразі створюється масив екземплярів типу Task, при ініціалізації вказується кількість елементів в масиві - їх 3, так як різновидів звітів теж 3 (UAL, HW, TUN). Кожен з трьох звітів ініціалізується лямбда виразом і запускається на виконання. Для кожного потоку буде паралельно виконуватися обробка своїх каналів, в залежності від типу звіту. Щоб отримати дані з усіх трьох потоків, потрібно почекати їх виконання, для цього викликається статичний метод WaitAll в класі Task, який приймає в якості агрументу масив екземплярів Task, які потрібно чекати. У нього передається раніше створений масив з екземплярами для звітів. У кожному методі refineChannels (для кожного виду звіту) створюється масив екземплярів класу Task, в якому при ініціалізації вказується кількість елементів, рівній кількості каналів в поточному звіті. У циклі, який проходиться по кожному каналу, ініціалізується кожен елемент масиву лямбда виразом, в якому прописана логіка парсинга одного каналу (рис. 3.3).

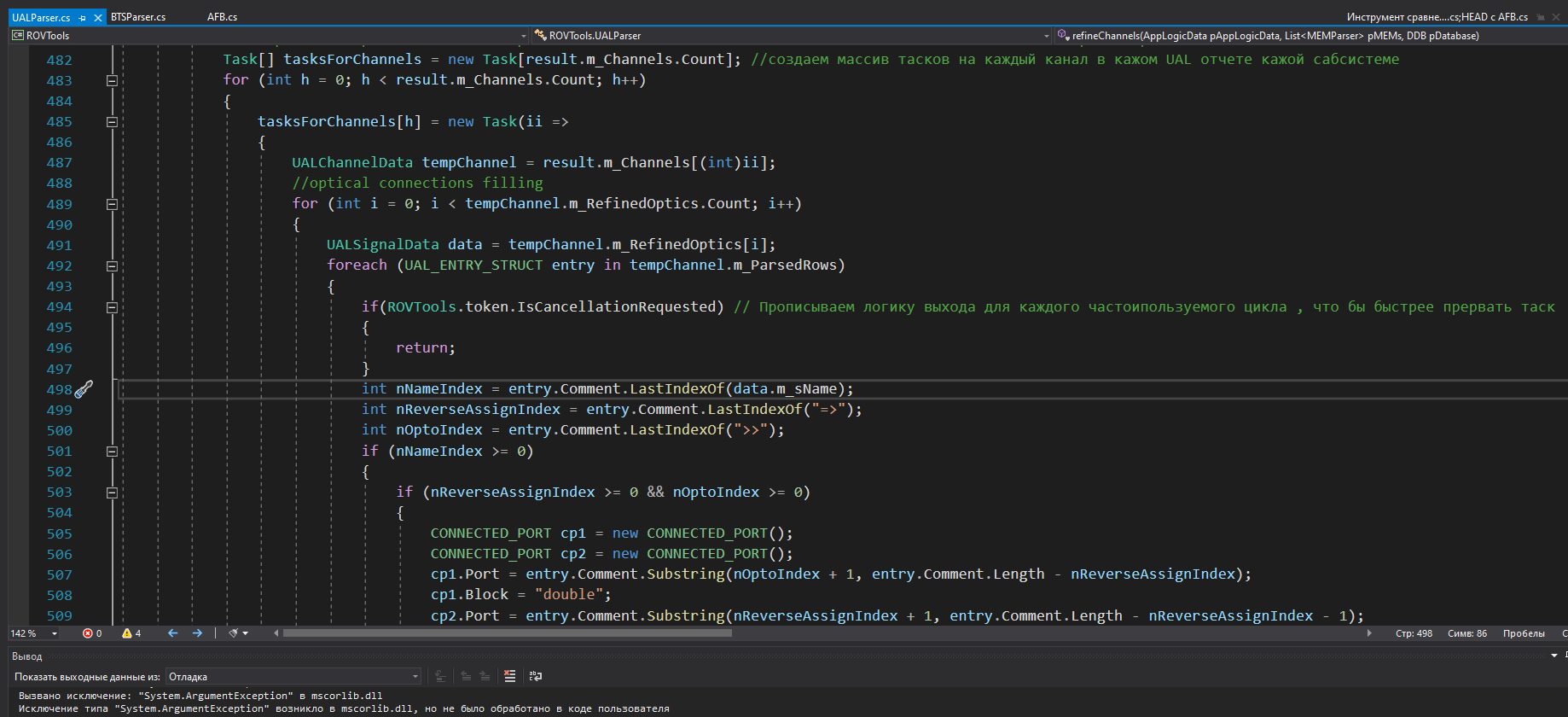


Рисунок 3.3

Так як існує можливість прервати побудову звіту в будь-яку секунду, потрібно реалізувати цей функціонал в кожному зі створених потоків. Для цього створюється статичне поле в класі ROVTools, яке зберігає токен скасування для потоків. І коли властивість структури CancellationRequested буде перебувати в стані true, треба повернутися з методу, який виконується потоком. Для цього у всіх критичних місцях, наприклад, всередині циклів або на початку методів, потрібно писати таку конструкцію:

if (ROVTools.token.IsCancellationRequested)

{

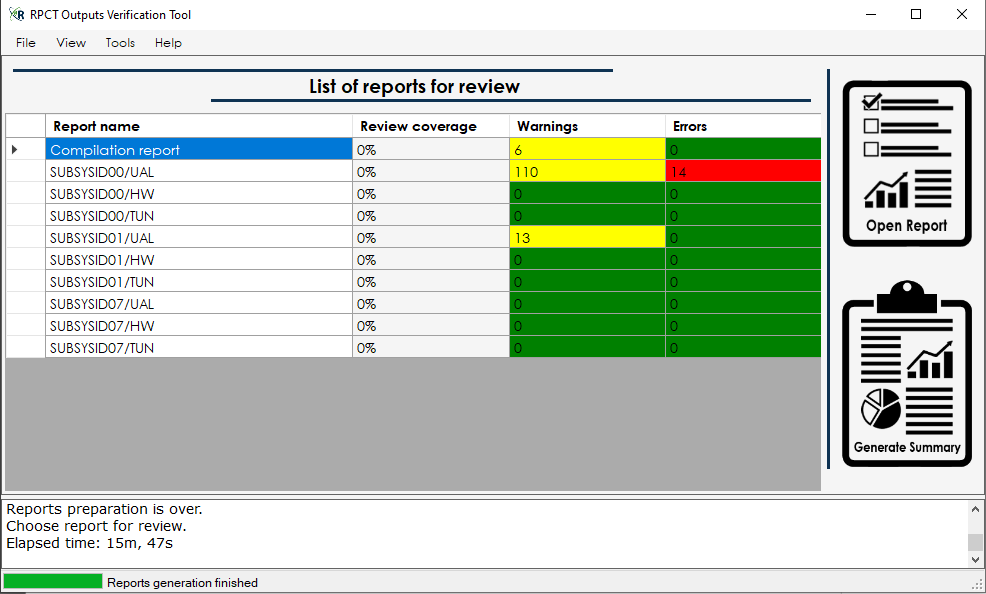
return;

}

CLR припинить виконання коду в методі і повернеться на рівень вище, в слідстві чого дозволить припинити виконувати потік і повернеться в точку виконання потоку.

## 3.3 Оцінка ефективності реалізованих засобів підвищення ефективності ROVT.

Для того щоб провести аналіз ефективності оптимізації роботи. Був здійснений запуск ROVT на найбільшому проєкті, який може перевірятися в ROVT. Витрачений час на генерацію звітів записаний внизу, в консолі. Запускаємо цей проєкт в версії без оптимізації та отримуємо такі дані – 15 хв, 47 сек.

 Рисунок 3.4 – Згенерований звіт без оптимізації

Після цього запускаємо проєкт із здійсненою оптимізацією за допомогою багатопоточності. Отримуємо такі дані – 29 сек.

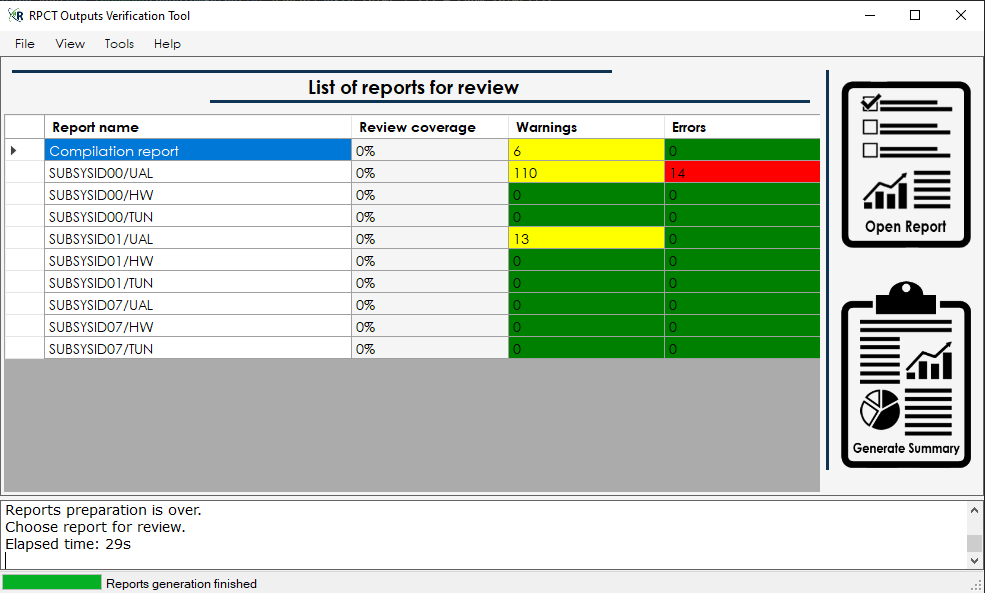


Рисунок 3.5 – Згенерований звіт із оптимізацією

# ВИСНОВКИ

Мета дипломної роботи полягала в розробці інструментів підвищення ефективності застосування програмних засобів автоматизованої верифікації результатів компіляції програмних проєктів критичних ІУС. Був розглянутий програмно-технічний комплекс з використанням платформи RadICS та наведені його види, побудова та його основні функції. ПТК на даний момент є актуальною системою для використання в атомній промисловості. Огляд основних процедур верифікації прикладної логіки спецплатформи RadiCS дозволяє вчасно вжити коригувальні та профілактичні заходи з метою усунення виявлених невідповідностей та, відповідно, уникнення або мінімізації претензій із боку зовнішніх та внутрішніх споживачів, покращення умов експлуатації та використання предмету огляду. Radiy та RadICS надають набір інструментів верифікації, які дозволяють виконувати надійну верифікацію AL, розробленої з RPCT. Розглянуто ROVT, як інструмент верифікації та його основний функціонал. Було проаналізовано недоліки поточної версії ROVT. Наведені декілька пропозицій щодо вирішення проблеми з продуктивністю програми та проаналізовано кожну з них також проаналізовано основні механізми реалізації багатопоточності в мові програмування С#, наведено можливі реалізації багатопоточності в платформі .Net. Була реалізована багатопоточность в програмному забезпеченні ROVT. При оцінюванні ефективності імплементованої оптимізації, з’ясовано, що продуктивність програмного забезпечення зросла в декілька разів. Отже, виконана робота повністю відповідає вимогам розробки та покращує поточну версію ROVT.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IEC 61508-2010. Functional safety of electrical/ electronic/ programmable electronic safety-related systems – Part 3: Software requirements.
2. IEC 61508-2010. Functional safety of electrical/ electronic/ programmable electronic safety-related systems – Part 7: Overview of techniques and measure
3. ПЛК [Електронний ресурс] – Режим доступу до ресурсу: <http://surl.li/xaak>.
4. Платформа RadICS [Електронний ресурс] – Режим доступу до ресурсу: <http://www.radiy.com/ru/>.
5. Программно-технический комплекс автоматического регулирования, разгрузки и ограниченой мощности реактора и ускоренной предупредительной защиты [Електронний ресурс] – Режим доступу до ресурсу: <http://www.radiy.com/ru/produktsiya-dlya-aes/produktsiya/sistemy-bezopasnosti-na-baze-fpga-dlya-aes/ptk-arm-rom-upz.html>.
6. Striuk O. User-Designed Application Logic Verification Approach and Tools [Електронний ресурс] / O. Striuk, V. Shamanskyi – Режим доступу до ресурсу: <https://radics.tech/user-designed-application-logic-verification-approach-and-tools/>.
7. Striuk O. Verification of Application Logic designed with Radiy Platform Configuration Tool [Електронний ресурс] / O. Striuk, V. Shamanskyi – Режим доступу до ресурсу: <https://radics.tech/verification-of-application-logic-designed-with-radiy-platform-configuration-tool/>.
8. Рекомендации по работе с потоками [Електронний ресурс] // Microsoft. – 2018. – Режим доступу до ресурсу: <https://docs.microsoft.com/ru-ru/dotnet/standard/threading/managed-threading-best-practices>.
9. Task Класс [Електронний ресурс] // Microsoft – Режим доступу до ресурсу: <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.tasks.task?view=net-5.0>.
10. ThreadPool Класс [Електронний ресурс] // Microsoft – Режим доступу до ресурсу:<https://docs.microsoft.com/ru/dotnet/api/system.threading.threadpool?view=net-5.0>.
11. ГОСТ 19.005-85. ЕСПД. Р-схемы алгоритмов и программ. Обозначения условные графические и правила выполнения.
12. ГОСТ 19.101-77. ЕСПД. Виды программ и программных документов.
13. ГОСТ 19.102-77. ЕСПД. Стадии разработки.
14. Ястребенецький М.О., Розен Ю.В., Виноградська С.В. та ін. Безпека атомних станцій. Система керування й захисту ядерних реакторів. Російською мовою.- К: ТОВ «Основа Принт», 2011.-С. 32-128.