МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра компьютерных технологий

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Масеевского Антона Михайловича

Тема работы:

РАЗРАБОТКА ВЕБ-ТЕХНОЛОГИИ ВИРТУАЛИЗАЦИИ ПЛК СРЕДСТВАМИ PYTHON-ИНТЕРПРЕТАТОРА ДЛЯ ИСПОЛНЕНИЯ POST-ПРОГРАММ

«К защите допущена»
Зав. КафКТ ФИТ НГУ,
д.т.н., доцент
Зюбин В. Е. /…………..
(ФИО) / (подпись)
20 мая 2023г.

Руководитель ВКР
Зав. КафКТ ФИТ НГУ,
д.т.н., доцент
Зюбин В. Е. /………...
(ФИО) / (подпись)
20 мая 2023г.

Соруководитель ВКР
Ассистент КИСТ ВКИ,
Башев В. И. /…………..
(ФИО) / (подпись)
20 мая 2023г.

Новосибирск, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра компьютерных технологий
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Зюбин В.Е.
(фамилия, И., О.)
………………….
(подпись)
23 ноября 2022г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Масеевскому Антону Михайловичу, группы 19203
(фамилия, имя, отчество, номер группы)

Тема: Разработка веб-технологии виртуализации ПЛК средствами Python-интерпретатора для исполнения poST-программ
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 23 ноября 2022г. №0377

Срок сдачи студентом готовой работы 20 мая 2023 г.

Исходные данные (или цель работы): Разработать прототип веб-технологии виртуализации ПЛК для целей процесс-ориентированного программирования на языке poST.

Структурные части работы:

Анализ предметной области, разработка технологии виртуализации ПЛК, имплементация и тестирование виртуального ПЛК на веб-сервере и тестирование его работы на модельной задаче.

Консультанты по разделам ВКР (при необходимости, с указанием разделов)

Руководитель ВКР
Зав. КафКТ ФИТ НГУ,
д.т.н., доцент
Зюбин В. Е. /…………..
(ФИО) / (подпись)
23 ноября 2022г.

Задание принял к исполнению
Масеевский А. М. /…………..
(ФИО студента) / (подпись)
23 ноября 2022г.

Соруководитель ВКР
Ассистент КИСТ ВКИ,
Башев В. И. /…………..
(ФИО) / (подпись)
23 ноября 2022г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 5

ВВЕДЕНИЕ 6

ГЛАВА 1. Анализ специфики разработки программ для ПЛК 9

1.1 Особенности задач промышленной автоматизации 9

1.2 Подходы к организации практических занятий при обучении программированию 10

1.2.1 Удалённые 10

1.2.2 При обучении программированию управляющих систем 11

1.3 Подходы к разработке программ для ПЛК 12

1.4 Процесс-ориентированный подход и язык poST 13

1.5 Требования к реализации, выявленные в результате анализа 15

ГЛАВА 2. Разработка виртуального ПЛК 17

2.1 Архитектура виртуального ПЛК 17

2.2 Правила трансляции poST в программу на языке Python 18

2.3 Реализация генератора в язык Python 26

2.4 Реализация транслятора 27

ГЛАВА 3. Имплементация виртуального ПЛК в веб-приложении и тестирование его работы на модельной задаче 29

3.1 Реализация веб-приложения 29

3.1.1 Пользовательский интерфейс 29

3.1.2 Веб-сервер 31

3.2 Тестирование работы виртуального ПЛК на модельной задаче 33

3.2.1 Реализация алгоритма на языке poST 34

3.2.2 Анализ полученных результатов 35

ЗАКЛЮЧЕНИЕ 37

СПИСОК ЛИТЕРАТУРЫ.. 39

ПРИЛОЖЕНИЕ А 42

ПРИЛОЖЕНИЕ Б.. 45

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПЛК – программируемый логический контроллер.

ОУ – объект управления.

АУ – алгоритм управления.

poST – process-oriented Structured Text.

ЛС – лабораторный стенд.

POU – Program Organization Unit, программный компонент.

ФБ – функциональный блок.

ИАиЭ – Институт Автоматики и Электрометрии.

СО РАН – Сибирское Отделение Российской Академии Наук.

ST – Structured Text, язык из состава IEC 61131-3.

AST – Abstract Syntax Tree, абстрактное синтаксическое дерево.

IDE – Integrated Development Environment, интегрированная среда разработки.

ВВЕДЕНИЕ

Качество образования, получаемого студентом, определяется не только уровнем теоретической подготовки, но и умением применять эти знания на практике. Такие навыки приобретаются через лабораторные работы и семинарские занятия. В настоящее время происходит активное введение дистанционных и компьютерных технологий в образовательный процесс. Особенно это актуально для программирования, ввиду отсутствия разницы между результатом очной и заочной практики, если есть подходящие для этого инструменты.

При изучении языков программирования для создания управляющих алгоритмов, также требуются практические занятия, более того, качественная подготовка даже важнее, чем при работе с языками высокого уровня, где есть возможность работы с абстрактными объектами и структурами данных, потому что, в зависимости от объекта управления, ошибка в алгоритме управления может повлечь большие финансовые потери, представлять опасность жизни и здоровью людей и даже вызвать экологические проблемы. Однако для этого требуется специфическое оборудование – программируемые логические контроллеры (ПЛК) и лабораторные стенды, реализующие в упрощённом виде некие объекты управления (ОУ). Требуются значительные финансовые затраты на их приобретение или создание и поддержку в рабочем состоянии. Кроме того, лабораторные стенды могут занимать много места и выходить из строя при ошибках обучающихся. Это означает, что одновременно проверять работоспособность своего алгоритма управления (АУ) в состоянии только ограниченное количество студентов, что может затягивать процесс обучения. Всё это создаёт потребность в программном симуляторе ПЛК, и, соответственно, разработка решения на базе интерпретатора подходящего языка.

Ранее для этой цели могли использоваться возможности специализированных программных пакетов и сред разработки для языков стандарта IEC 61131-3 [1], например CODESYS [5] или LabView [2], но, в связи

с развитием облачных технологий и введённых региональных ограничений, данный метод постепенно теряет свою актуальность.

Цель дипломной работы – разработать прототип веб-технологии виртуализации ПЛК для целей процесс-ориентированного программирования на языке poST.

Для достижения этой цели были поставлены следующие задачи:

1. Анализ особенностей промышленной автоматизации, подходов к программированию ПЛК, организации практических занятий при обучении программированию.
2. Формирование списка требований к технологии виртуализации.
3. Разработка архитектуры симулятора ПЛК на основе интерпретатора Python.
4. Определение правил перевода poST-программ в язык интерпретатора, используемого для симуляции ПЛК.
5. Разработка транслятора языка poST, генерирующего код на языке используемого интерпретатора.
6. Имплементация симулятора ПЛК в виде веб-приложения.
7. Исследование применимости разработанных программных средств на модельной задаче.

Новизна данной работы состоит в создании веб-технологии виртуализации ПЛК, в которой сгенерированный из poST-программы код на языке Python загружается в веб-приложение – симулятор ПЛК, представляющий из себя интерфейсную обертку и интерпретатор Python, реализующие, в свою очередь, модель гиперпроцесса.

Практическая ценность работы по итогам данной работы получено веб-приложение – симулятор ПЛК, в котором реализованы средства запуска программы на языке poST при помощи транслятора и интерпретатора языка Python, которое можно использовать для запуска, проверки корректности работы и тестирования алгоритмов управления для программируемых логических контроллеров.

Структура и объем работы. Работа состоит из введения, 3 глав и заключения. В первой главе рассматриваются особенности ПЛК, существующие подходы к организации практических занятий по программированию, модель гиперпроцесса для программ на ПЛК, проводится их анализ и формируется список требований к разрабатываемому подходу на основе его результатов. Во второй главе описан процесс разработки технологии виртуализации ПЛК с учётом сформулированных в первой главе требований. Третья глава посвящена имплементации виртуального ПЛК в веб-приложении и тестированию технологии на модельной задаче.

ГЛАВА 1. Анализ специфики разработки программ для ПЛК

1.1 Особенности задач промышленной автоматизации

Практически вся промышленная автоматизация реализуется на цифровых системах управления. Ядром такой системы является программируемый логический контроллер. ПЛК могут применяться для широкого ряда объектов управления: от домашних приборов до требовательных к безопасности инфраструктурно важных объектов. Они представляют собой микропроцессорный модуль с множеством входов и выходов.

Самый распространённый метод применения – реализация некоторой производственной технологии. Через входы поступают сигналы датчиков о текущем состоянии объекта управления и, возможно, сигналы с рабочего места оператора, в зависимости от них выводятся сигналы для механизмов объекта управления в режиме реального времени. Эта особенность обеспечивается при помощи цикличности алгоритма управления по схеме: “считывание сигналов датчиков и ввода оператора” - “обработка” - “формирование и выдача на их основе выходных сигналов для объекта управления”, данный цикл называется циклом сканирования [9].

В большинстве случаев объекты управления предполагают множество параллельных процессов. События, происходящие в разных его компонентах, возникают независимо и без определённой последовательности. Чтобы избежать комбинаторного перебора различных комбинаций показаний датчиков и ввода оператора, алгоритмы управления выстраиваются из множества независимых или слабозависимых частей. Это свойство зовётся логическим параллелизмом [12]. Логический параллелизм подкрепляется и физическим за счёт множества выходов ПЛК на различные компоненты ОУ.

Объект управления является физической системой. Это значит, что он подчиняется законам естественных наук. Для некоторых процессов требуется время, например, насосы не могут выкачать всю жидкость мгновенно. То есть для алгоритма управления необходима синхронизация исполнения с процессами

внешней среды, в связи с чем требуется работа с временными интервалами: задержками, паузами, тайм-аутами.

1.2 Подходы к организации практических занятий при обучении программированию

1.2.1 Удалённые

На данный момент одним из самых распространённых методов организации практических занятий по программированию является выдача заданий для самостоятельной реализации. Такой подход хорошо подходит, если не предполагается использование специфического оборудования или можно работать с абстрактными объектами и структурами данных. Но для эффективности этого метода у обучающегося должны быть средства для самостоятельной проверки работоспособности написанной программы.

Другим методом является загрузка кода программы на сервер [20], где она исполняется и её элементы должны пройти серию тестов. Таким образом можно проверить правильность работы алгоритма, его скорость, использование памяти и, благодаря тому что проверяющая система имеет исходный код программы, проверить некоторые дополнительные условия, например, ограничить применение средств языка, которые для данной задачи можно считать ошибочными.

Если же требуется какое-то оборудование, то единственным способом является его предоставление студенту или наличие какого-либо программного имитатора. Для задач промышленной автоматизации это означает установку программного пакета, например, Rockwell software RSLogix 500 [19] или CODESYS [18], в котором, помимо алгоритма, необходимо описывать и объект управления, если такая возможность предусмотрена. Кроме того, для использования в образовательных целях может требоваться лицензия или распространение может быть регионально ограничено.

1.2.2 При обучении программированию управляющих систем

1.2.2.1 Реальные объекты управления

При использовании реального объекта управления в качестве площадки для обучения и отладки наглядно видно работу алгоритма управления и вносимых в него изменений. Но у подхода есть значительный недостаток – опасность возникновения аварии, если студент допустил ошибку. В зависимости от объекта управления это может привести к серьёзным финансовым потерям, представлять опасность для жизни и здоровья людей и даже привести к экологической катастрофе. Помимо этого, очень сложно изменить условия среды для тестирования работы программы, а имитацию отказов придётся проводить, вызывая намеренный отказ оборудования, если не предусмотрены способы не аварийного отключения отдельных элементов системы. Модификация в целях обучения как правило невозможна, если вообще целесообразно модифицировать промышленный объект для таких целей. Подход исключает пошаговое исполнение алгоритма управления для отладки. Вывод показаний датчиков как правило происходит на интерфейс автоматизированного рабочего места оператора.

1.2.2.2 Физические имитаторы

Физические имитаторы, они же лабораторный стенд (ЛС), – моделирование объекта управления с использованием лабораторного физического оборудования. По сравнению с использованием реального оборудования появляется возможность предусмотреть изменение среды тестирования или имитировать отказы оборудования. Как и предыдущий вариант, не предусматривает пошаговое исполнение алгоритма.

Такой стенд требует затрат на его создание и использование, а в зависимости от задачи ещё и места, этими факторами осложнено его тиражирование. По сравнению с реальными объектами лабораторные стенды относительно безопасны, но всё равно могут требовать соблюдения мер предосторожности и техники безопасности. Кроме того, стенды не застрахованы

от ошибок студентов при написании алгоритма или от износа оборудования, поэтому требуют постоянного технического обслуживания.

1.2.2.3 Программно-аппаратное моделирование

Лабораторный стенд в этом случае представляет собой набор контроллеров [4], часть моделируемой системы может быть представлена подключением физических объектов, например, устройств управления.

Это может быть как проприетарный контроллер с заранее заданным набором задач, так и собственная разработка. При собственной разработке значительно повышается возможность модификации модели объекта управления. Изменение внешних условий или имитация отказа оборудования могут быть предусмотрены в программе или при помощи отключения внешних устройств.

1.2.2.4 Программное моделирование

Все предыдущие методы требовали, как минимум, ПЛК и некоторое внешнее оборудование. Данный подход подразумевает, что как поведение ПЛК, как и лабораторный стенд, представляют собой программный продукт [3]. Отсюда следует, что любое изменение или модификация – вопрос только разработки программного обеспечения и его обновления. Не требуется дополнительного оборудования, кроме того, которое есть в компьютерном классе. Возможно пошаговое исполнение программы и вывод отладочной информации.

1.3 Подходы к разработке программ для ПЛК

На данный момент основным методом разработки программ для ПЛК является использование языков стандарта IEC 61131-3 [1]: ассемблероподобного Instruction List, паскалеподобного Structured Text или одного из графических языков: Ladder Diagram, Function Block Diagram, Sequential Function Chart.

Программа на языке стандарта состоит из ресурсов, которые группируются в системе управления ПЛК, предоставляющей для них возможность обмена данными. Сам ресурс может исполнять одну или несколько задач. Задачи, в свою очередь, выполняют один или несколько программных компонентов (Program

Organization Unit – POU), ими являются функция, функциональный блок и программа.

Объявление POU содержит в себе объявление физических или логических входных и выходных переменных с указанными типами.

Существуют также объектно-ориентированные расширения стандарта IEC 61131-3 [11]. Они заключаются в добавлении поддержки интерфейсов, определения класса на основе функционального блока с его расширением методами и возможностью наследования. Однако из-за отсутствия стандарта каждый производитель ПЛК или сред разработки может добавлять элементы объектно-ориентированного программирования со своей собственной реализацией, что вредит переносимости программ.

IEC 61499 [8] был создан на основе IEC 61131-3. Он был разработан для создания распределённых алгоритмов управления для взаимодействующих систем. Одно из основных отличий этого стандарта – отход от циклической модели, в которой порядок исполнения процессов не был чётко определён, так что реализация зависела от производителя ПЛК или разработчика среды разработки, из-за чего страдала переносимость, и переход к событийной, где события в одном блоке запускают другой. Всё это достигается введением нескольких видов функциональных блоков (ФБ): базисного ФБ, составного ФБ и сервисного интерфейсного ФБ, а также появлением у них интерфейсов с событийными и информационными входами и выходами. Базисные ФБ представляют собой машину состояний, с состояниями могут быть связаны выходные события и алгоритмы. Функциональность составного ФБ определяется сетью входящих в него базисных ФБ. Сервисные интерфейсные ФБ по сути являются обёрткой над аппаратными средствами.

1.4 Процесс-ориентированный подход и язык poST

Основы процесс-ориентированного подхода для разработки алгоритмов автоматизации киберфизических системы были разработаны в Институте Автоматики и Электрометрии СО РАН (ИАиЭ СО РАН). Основная идея этой

парадигмы в добавлении нового уровня абстракции для языков общего назначения, а именно процессов и состояний [17].

Центральное определение парадигмы – гиперпроцесс [12]. Это множество взаимодействующих процессов с общей памятью и средствами синхронизации исполнения.

Процесс – это расширенный конечный автомат, что роднит его с базисными ФБ стандарта IEC 61499. К конечному автомату добавлены остановочные состояния, возможность контролировать состояния других процессов и средства синхронизации исполнения.

Состояние – это состояние процесса с набором действий, описанных в алгоритме и набором операторов:

1. Переход процесса в другое состояние.
2. Запуск, остановка или остановка по ошибке текущего или другого процесса. Если процесс остановился, то вывести его из этого состояния может только другой процесс.
3. Проверки состояний текущего процесса на активность, остановку и причину остановки: нормальная или по ошибке.
4. Таймаут – временной контроль нахождения процесса в данном состоянии. Одно из средств синхронизации состояний. Вторым является сохранение времени начала работы состояния и возможность его сброса.

Состояния могут быть замкнутыми.

Программы реализуются циклом, на каждой итерации которого процессы выполняют одно из своих состояний или простаивают, если находятся в состоянии остановки или остановки по ошибке. В начале исполнения инициализирующий процесс находится в одном из своих состояний. Остальные процессы находятся в состоянии нормальной остановки.

Одним из средств реализации алгоритмов в процесс-ориентированной парадигме является разработанный в ИАиЭ СО РАН язык poST [6]. poST – это процесс-ориентированное расширение паскалеподобного языка Structured Text (ST) стандарта IEC 61131-3. Он предназначен для разработки программ для ПЛК

и решение задач промышленной автоматизации. Реализован транслятор в ST, что позволяет программам, написанным на poST, работать на существующих системах, поддерживающих данный стандарт, но из-за нового уровня абстракции упрощается поддержка программ. Кроме того, язык поддерживает шаблоны, что позволяет избавиться от дублирования кода для повторяющихся элементов объекта управления.

Из-за того, что этот язык уже разработан с учётом особенностей работы ПЛК и, благодаря наличию транслятора в программу на языке ST, может быть интегрирован в существующие системы, поддерживающие стандарт IEC 61131-3, а значит написанные на нём алгоритмы можно использовать в прикладных задачах, он был выбран в качестве языка реализации алгоритма управления для виртуального ПЛК. Таким образом, понадобится реализация транслятора в программу на языке интерпретатора, который будет реализовывать концепцию гиперпроцесса.

1.5 Требования к реализации, выявленные в результате анализа

В результате анализа выявлено, что удалённый виртуальный ПЛК должен объединить в себе методы удалённого обучения программированию, программного моделирования поведения ПЛК для выполнения лабораторных работ, сохранив его особенности.

Итоговый список требований, возникших в результате анализа, получился следующим:

- автоматический режим управления на основе алгоритма управления и ручной режим для имитации автоматизированного рабочего места оператора и его действий;
- виртуальный ПЛК должен отрабатывать программу циклически по схеме “считывание сигналов датчиков и ввода оператора” - “обработка” - “формирование и выдача на их основе выходных сигналов для виртуального объекта управления”;

- должна быть сохранена семантика языка poST, в том числе при работе с временными интервалами: задержками, паузами, тайм-аутами; логическим параллелизмом и др.;
- должна быть предоставлена отладочная информация о состоянии процессов, значения счетчиков времени, значения переменных;
- в ручном режиме должен быть предоставлен механизм получения исходной poST-программы для изменения алгоритма управления.

ГЛАВА 2. Разработка виртуального ПЛК

2.1 Архитектура виртуального ПЛК

Алгоритм работы ПЛК описан отдельной программой, в которой создаётся экземпляр программы алгоритма управления, реализующей модель гиперпроцесса, после чего запускается цикл сканирования с необходимой периодичностью. Можно поставить симулятор на паузу или остановить его работу.

В качестве интерпретатора для симулятора ПЛК был выбран Python. Выбор обусловлен простотой запуска скриптов, написанных на этом языке, без необходимости компилировать проект заново, что полезно при запуске разных программ. В теории возможна реализация и на компилируемых языках без повторной компиляции, с запуском программы через системный вызов, но тогда работу по считыванию и записи данных придётся перенести в исполняемый код алгоритма управления, что нарушает поведенческую модель ПЛК, а значит и одно из предъявляемых требований, и требует реализации отдельного генератора, который предусматривает такую модификацию работы алгоритма управления.

Виртуальный ПЛК должен исполнять алгоритм управления, описанный на языке poST, для этого требуется преобразовать его в язык интерпретатора. Для разработки транслятора из языка poST в язык Python было принято решение модифицировать уже разработанный в ИАиЭ СО РАН транслятор в язык ST [7].

На вход симулятор принимает путь, по которому будут создаваться файлы для вывода информации о его работе и считывания значений входных переменных и флагов постановки на паузу или остановки работы симулятора. Перед запуском цикла сканирования происходит инстанцирование класса программы, имена и типы её входных переменных записываются в файл “inputs” для того, чтобы внешние программы, например, веб-интерфейс, могли получить об этом информацию и, возможно, присвоить новые значения. В момент считывания данных в цикле сканирования значения входных переменных считываются из файла “sim_in” и записываются в программу до старта

следующей итерации исполнения. Похожим образом реализовано считывание значений флагов постановки на паузу или остановки работы симулятора, которые на данный момент являются единственными поддерживаемыми внешними командами, призванные имитировать деятельность оператора. В состоянии паузы цикл сканирования уменьшается до считывания входных переменных, чтобы получить информацию о том, когда пауза будет снята. Общее время нахождения в состоянии паузы вычитается из счётчика времени. После выполнения итерации программы значения времени процессов и значения переменных, отсортированные по типу, записываются в файл “outputs”. Частота исполнения цикла сканирования задаётся в исходной poST-программе. Полный алгоритм исполнения логики ПЛК находится в приложении А.

Принцип работы виртуального ПЛК приведён на рисунке 1.

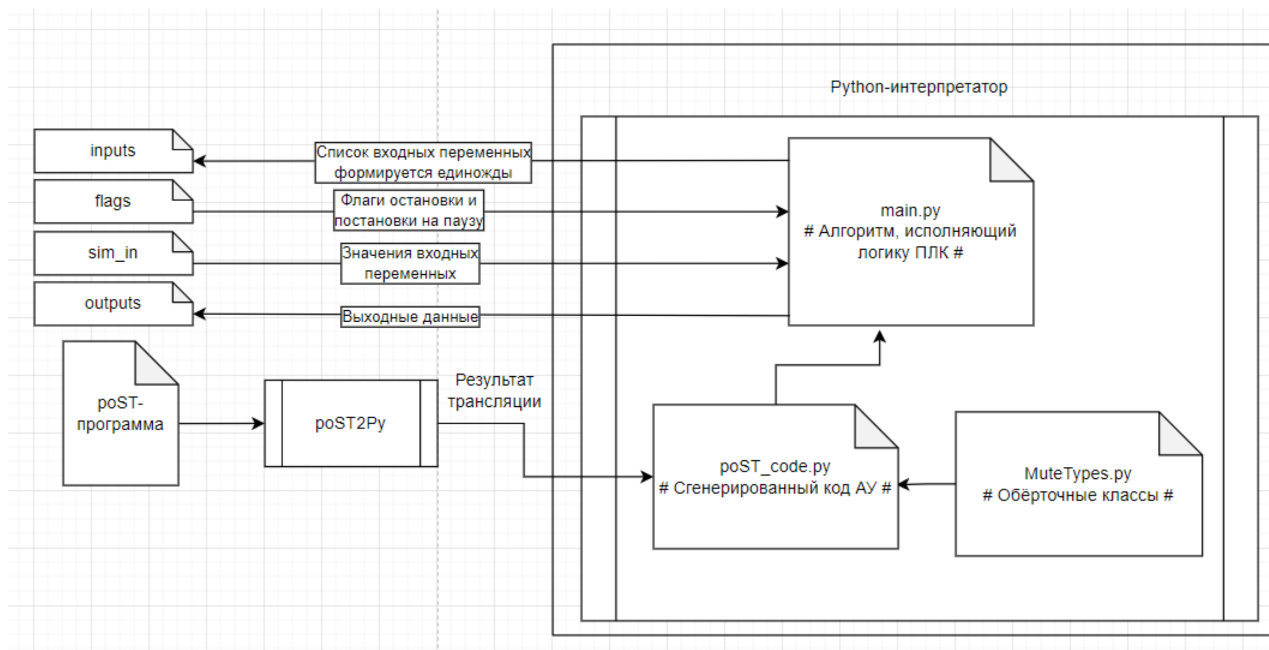

Рисунок 1 – Принцип работы виртуального ПЛК

2.2 Правила трансляции poST в программу на языке Python

Основной элемент гиперпроцесса – программа, реализуется как класс, экземпляр которого запускается из основного класса симулятора, реализующего логику контроллера. Программа хранит в себе поле, которое является списком со экземплярами процессов данной программы, которые выполняются по очереди в зависимости от своего состояния. Его полным аналогом, согласно

грамматике языка poST, является функциональный блок, с тем лишь исключением, что он не используется как точка входа и, как правило, является подпрограммой.

Процессы переводятся в эквивалентные по функционалу классы, которые сами по себе являются подклассами класса программы. Каждый процесс имеет в себе перечисление состояний, в котором, помимо состояний самого процесса, находятся состояния Stop и Error, текущее состояние, функцию исполнения, являющуюся машиной состояний данного процесса.

Все числовые и строковые литералы преобразуются в реализованные обёрточные классы эквивалентные типы данных, кроме констант, по причине их отсутствия в языке Python, поэтому они преобразуются в обычные типы данных. Другими исключением являются шестнадцатеричные, восьмеричные, двоичные числа и битовые строки, которые в языке poST имеют собственные литералы, в Python они переводятся с помощью соответствующих предикатов.

Литерал времени, использующийся для задания временных интервалов, преобразуется в семантически идентичную конструкцию timedelta из модуля datetime.

Массивы переводятся в объекты типа list, указание их длины и типов элементов внутри при объявлении отбрасывается.

Циклы переводятся в эквивалентные циклы языка Python, за исключением цикла REPEAT-UNTIL, в Python отсутствует аналог цикла do-while, так что трансляция выглядит как цикл while True с условием для выхода с помощью команды break.

Из ветвлений стоит обратить внимание на CASE OF. Именно он служит основой машины состояний для процессов. Семантически эквивалентное ветвление match-case появилось в Python 3.10 [22], и именно из-за этого 3.10 является выбранной версией интерпретатора.

Арифметические и логические операторы транслируются в семантически эквивалентные операторы.

Переменные в языке poST сохраняются по ссылке, чтобы изменение значения переменной было видно и в массивах, которые их хранят, это полезно для хранения значений однотипных процессов, заданных, например, в конфигурации программы на языке poST. Python имеет несколько неизменяемых типов данных, для которых создаётся новый адрес в памяти при создании и присваивании [21], из-за чего отдельно понадобилась реализация обёрточных классов со всеми основными операторами, используемыми в итоговом алгоритме управления. Python не позволяет переопределять присваивание, так как динамическая типизация – одна из основных особенностей языка, поэтому присваивание новых значений для обёрточных классов осуществляется при помощи метода __set__, которому передаётся значение базового класса или другой объект обёрточного класса.

Кроме того, переменные в языке poST делятся на: входные, выходные, глобальные, внешние, временные и обычные. Для сохранения этого разделения для целей верификации переменные рассортированы в соответствующие словари. Отдельно следует упомянуть переменные шаблонных процессов. Дело в том, что входные и выходные переменные отсутствуют в сигнатуре шаблонного программного элемента. Поэтому появилась необходимость в составе транслятора иметь парсер текста конфигурации, который выделяет входные переменные для шаблонных процессов, объявленных в ней.

Операторы RESTART / START PROCESS <процесс>, STOP / STOP PROCESS <процесс> и ERROR / ERROR PROCESS <процесс> реализованы при помощи вызовов соответствующих сгенерированных функций, параметром которых является процесс, над которым проводится операция запуска, остановки или перехода к следующему состоянию. Функция start(p) переводит процесс p в его изначальное состояние, функция stop(p) – в состояние остановки, error(p) – в состояние остановки по ошибке.

Оператор SET STATE <состояние> преобразуется в функцию set_state(state) класса процесса, аргументом которой является состояние, в которое необходимо перевести данный процесс. Оператор SET NEXT

преобразуется в похожую функцию set_next(), которая переводит процесс в следующее состояние, либо в вызывает функцию start, если текущее состояние последнее, не являющееся состоянием остановки или остановки по ошибке.

Оператор PROCESS <процесс> IN STATE <ключевое слово> в сравнение текущего состояния процесса с состояниями Stop и Error, преобразование отличается в зависимости от ключевого слова: ACTIVE – проверка, что процесс не находится в состоянии остановки или остановки по ошибке, INACTIVE – наоборот, ключевые слова STOP и ERROR проверяют, находится ли процесс в состоянии остановки соответствующего вида.

Оператор RESET TIMER обновляет время старта текущего процесса.

Оператор TIMEOUT <литерал времени> преобразуется в логическое выражение, в котором разница между текущим временем и временем старта состояния сравнивается с возвращаемым значением total_seconds() объекта класса timedelta, в который преобразуется данный литерал времени.

Пример программы на языке poST и результат её трансляции приведены в листингах 1 и 2 соответственно.

```
PROGRAM program

    VAR_INPUT
        input1 : BOOL;
        input2 : BOOL;
    END_VAR

    VAR_OUTPUT
        output1 : BOOL;
        output2 : BOOL;
    END_VAR

    PROCESS process1
        STATE state1
            IF input1 THEN
                output1 := TRUE;
                SET NEXT;
            END_IF
        END_STATE

        STATE state2
            IF input1 THEN
                RESET TIMER;
            END_IF
```

```
            TIMEOUT T#2s THEN
                output1 := FALSE;
                SET STATE state1;
                START PROCESS process2;
            END_TIMEOUT
        END_STATE
    END_PROCESS

    PROCESS process2
        STATE state1
            IF input2 THEN
                output2 := TRUE;
                SET NEXT;
            END_IF
        END_STATE

        STATE state2
            IF input2 THEN
                RESET TIMER;
            END_IF
            TIMEOUT T#2s THEN
                output2 := FALSE;
                SET STATE state1;
            END_TIMEOUT
        END_STATE
    END_PROCESS

END_PROGRAM
```

Листинг 1 – Пример программы на языке poST

```
import time
import enum
from MuteTypes import MuteBool, MuteNum, MuteStr, MuteBytes,
get_value
from datetime import timedelta

# Словари для сортировки переменных по типу, в дальнейшем в
# классах объявление их области видимости приводиться в листинге
# не будет
inVars = {}
outVars = {}
inOutVars = {}
Vars = {}
exVars = {}
globVars = {}

# Для хранения состояний процессов
pStates = {}

# Для хранения последнего времени запуска процессов
pTimes = {}
_global_time = MuteNum(0.0)
```

```
Vars['_global_time'] = _global_time
_g_p_process1_time = MuteNum(0.0)
Vars['_g_p_process1_time'] = _g_p_process1_time
_g_p_process2_time = MuteNum(0.0)
Vars['_g_p_process2_time'] = _g_p_process2_time

# Частота исполнения программы
taskTime = timedelta()

# Входные переменные
input1 = MuteBool(False)
inVars['input1'] = input1
input2 = MuteBool(False)
inVars['input2'] = input2

# Выходные переменные
output1 = MuteBool(False)
outVars['output1'] = output1
output2 = MuteBool(False)
outVars['output2'] = output2

processesDict = {}


# Функции внешнего влияния на процессы
def start(p):
    p.curState = 1
    pStates[f'{p.name()}_state'] = p.States(p.curState).name


def stop(p):
    p.curState = 254
    pStates[f'{p.name()}_state'] = p.States(p.curState).name


def error(p):
    p.curState = 255
    pStates[f'{p.name()}_state'] = p.States(p.curState).name


# Программа
class Program:
    # Список процессов программы
    processes = []

    # Класс процесса process1
    class process1:
        startTime = ""
        # Класс состояний процесса process1
        class States(enum.Enum):
            state1 = enum.auto()
            state2 = enum.auto()
```

```
                Stop = 254
                Error = 255

            def set_state(state):
            self.curState = state.value
            pStates[f'{self.name()}_state'] =
self.States(self.curState).name

            def set_next(self):
            self.curState += 1
               pStates[f'{self.name()}_state'] =
self.States(self.curState).name

            def name(self):
            return self.__class__.__name__

            curState = States.Stop.value

     # Функция исполнения процесса, реализующая машину состояний
      def run(self):
          match self.curState:
               case self.States.state1.value:
                    # Время запуска процесса, находится здесь,
                    # чтобы не перезаписывалось в состояниях
                    # остановки
                    self.startTime = _global_time
                    pTimes['process1_time'] = self.startTime
                    if input1:
                         # Присванивание нового значения объекту
                         # обёрточного класса
                         setVariable('output1', MuteBool(True))
                         # SET NEXT
                         self.set_next()
               case self.States.state2.value:
                    self.startTime = _global_time
                    pTimes['process1_time'] = self.startTime
                    if input1:
                         # RESET TIMER
                         setVariable('_g_p_process1_time',
_global_time)
                    # TIMEOUT T#2s
                    if (_global_time - _g_p_process1_time) >=
timedelta(seconds=2).total_seconds():
                         setVariable('output1', MuteBool(False))
                         set_state(self.States.state1)
                         # START PROCESS process2
                         start(processesDict['process2'])
               case _:
                    pass

     # Создание экземпляра процесса
     process1 = process1()
```

```
    processes.append(process1)
    processesDict['process1'] = process1
    pStates['process1_state'] =
process1.States(process1.curState).name
    pTimes['process1_time'] = process1.startTime

    class process2:
        startTime = ""
        global pStates

        class States(enum.Enum):
        state1 = enum.auto()
        state2 = enum.auto()
        Stop = 254
        Error = 255

        def set_state(state):
        self.curState = state.value
        pStates[f'{self.name()}_state'] =
self.States(self.curState).name

        def set_next(self):
        self.curState += 1
        pStates[f'{self.name()}_state'] =
self.States(self.curState).name

        def name(self):
        return self.__class__.__name__

        curState = States.Stop.value

        def run(self):
        match self.curState:
            case self.States.state1.value:
                self.startTime = _global_time
                pTimes['process2_time'] = self.startTime
                if input2:
                    setVariable('output2', MuteBool(True))
                    self.set_next()
            case self.States.state2.value:
                self.startTime = _global_time
                pTimes['process2_time'] = self.startTime
                if input2:
                    setVariable('_g_p_process2_time',
_global_time)
                if (_global_time - _g_p_process2_time) >=
timedelta(seconds=2).total_seconds():
                    setVariable('output2', MuteBool(False))
                    set_state(self.States.state1)
            case _:
                pass
```

```
	process2 = process2()
	processes.append(process2)
	processesDict['process2'] = process2
	pStates['process2_state'] =
process2.States(process2.curState).name
	pTimes['process2_time'] = process2.startTime

	start(processes[0])

	def run_iter(self):
		global _global_time
		Vars['_global_time'] = _global_time
	for process in self.processes:
	process.run()


def setVariable(name, value):
	try:
		globals()[name].__set__(value)
	except AttributeError:
		globals()[name] = value
```

Листинг 2 – пример сгенерированной Python программы. Примечание: комментарии в коде представлены для наглядности и не генерируются транслятором.

Шаблонные процессы, объявляемые в конфигурации, преобразуются в классы процессов, с полным сохранением логики, описанной в шаблоне, имя класса процесса берётся из его объявления в конфигурации.

2.3 Реализация генератора в язык Python

Генерация кода осуществляется с помощью абстрактного синтаксического дерева (AST) [10], которое прошло этап валидации.

Для реализации генератора был модифицирован генератор в язык ST. Генератор повторяет структуру AST, но к каждому узлу был добавлен метод генерации текста, преобразующий данный узел в код на языке Python. Конструкторы таких классов принимают на вход оригинальные узлы программы, извлекают их них информацию и создают классы детей-генераторов, которые имеют свои методы генерации текста. Таким образом при добавлении в язык poST новых конструкций или модификации уже существующих в результате его развития и поддержки, понадобится реализовать только генераторы текста для новых узлов и, при необходимости, модифицировать генерацию уже существующих, кроме того, так как, по сути, генератор

представляет собой полную копию AST языка с дополнительным методом генерации текста, то это упрощает параллельную разработку языка и его транслятора.

Когда для генерации Python-кода нужна была дополнительная информация, то модифицировались только классы генератора, без модификации исходного дерева программы. Примерами таких добавлений могут служить дополнительные переменные, новые методы или изменения работы уже готовых. Так, например, для реализации преобразования временных литералов в объекты классов timedelta требовалось достать значения времени, которые задают временные промежутки в днях, часах, минутах, секундах и миллисекундах, этого не было в модифицируемом генераторе, так как синтаксис временных литералов был идентичен между языками poST и ST. Самым заметным таким дополнением стал парсер текста конфигурации программы для выявления входных переменных для целей их считывания, так как все переменные, объявленные в конфигурации, в языке poST считаются глобальными.

Первый этап генерации – инстанцирование дерева исходной программы на языке poST и формирование генераторов из его узлов. Второй – вызов метода generate у корня дерева генераторов, который работает по принципу обхода в глубину, с генерацией текста при возвращении от детей к родителям и соединения текста генераторов-детей со сгенерированным текстом родителя. При успехе генерации создаётся файл с расширением “.py”, в который записывается полученный текст.

2.4 Реализация транслятора

Реализация транслятора не претерпела особых изменений по сравнению с транслятором из языка poST в язык ST, заменён был только генератор.

Транслятор – это Java-программа в виде JAR архива, которая включает в себя ядро языка poST для проведения этапов синтаксических и семантических проверок, а также все необходимые для его работы библиотеки в JAR формате. Таким образом транслятор можно запускать на любом устройстве, на котором

установлена виртуальная машина Java при помощи команды “java -jar”. Транслятор принимает на вход путь до файла с расширением “.post”.

Для разработки и создания генератора кода на языке Python использовались средства Eclipse IDE. Генератор был реализован с использованием языка Xtend [16], а создание происходит путём копирования кода генератора в ядро языка poST, после чего Eclipse IDE генерирует из классов в формате Xtend Java-классы. Сборка происходит методом FAT JAR [13], благодаря чему в итоговый JAR-архив включаются все необходимые для работы внешние библиотеки.

ГЛАВА 3. Имплементация виртуального ПЛК в веб-приложении и тестирование его работы на модельной задаче

3.1 Реализация веб-приложения

За основу веб-приложения были взяты JAR транслятор языка poST в язык Python и симулятор ПЛК, написанный на языке Python. Обе программы запускаются как отдельные процессы операционной системы на веб-сервере, который отправляет пользователю результаты их работы по GET-запросу.

Веб-приложение должно поддерживать функцию загрузки исходного кода алгоритма управления на сервер, трансляцию исходного кода в Python-программу, запуск полученной программы в симуляторе ПЛК с возможностями ввода значений входных данных, отправки вывода обратно пользователю, постановки работы симулятора на паузу или остановки его работы. Помимо этого, также необходимо поддерживать пользовательских сессий для сохранения результатов работы пользователя и возвращения к тому же состоянию, из которого был произведён выход.

Для реализации серверной части было решено использовать веб фреймворк Flask [14], так как он имеет нативную реализацию сессий пользователей. Серверная часть запускает транслятор и симулятор как отдельные процессы системы, в случае транслятора дожидаясь результатов его работы, и как параллельный процесс в случае симулятора, с функциями считывания данных из создаваемых им файлов по POST-запросу клиента.

Для пользовательской части используются язык разметки HTML, стили CSS и язык JavaScript [15] для реализации логики на стороне клиента.

3.1.1 Пользовательский интерфейс

Пользовательский интерфейс реализован на одной странице. Слева находятся текстовые поля, которые реализуют необходимый для работы транслятора функционал: одно изменяемое для ввода текста исходной poST-программы и 2 неизменяемых: для вывода ошибок и предупреждений во время синтаксической и семантической проверок введённого кода и вывода сгенерированной программы на языке Python. Справа находятся элементы

необходимые для работы пользователя с симулятором и блок управления: текстовое поле вывода выходных данных работы симулятора, поле ввода значений входных переменных; блок управления содержит следующий набор кнопок: трансляция, открытие файла с пользовательского компьютера, скачивание poST кода, скачивание Python-кода, запуск симуляции, остановка симуляции, постановка выполнения алгоритма управления симулятором на паузу. Все кнопки отправляют соответствующий запрос на сервер, вызывая необходимую функцию.

Во время отправки POST-запроса на трансляцию poST-программы в её описание на языке Python, помимо идентификатора команды, отправляется текст из поля для ввода.

Из поля для ввода входных переменных во время исполнения работы симулятора формируется список переменных и их текущие значения и отправляются на сервер, где они записываются в файл, из которого симулятор считывает входные значения. В ответ на все запросы, совершаемые при помощи кнопок, кроме загрузки poST и Python кода с сервера на устройство пользователя, сервером отправляется новая веб-страница. Сервер автоматически заполняет текстовые поля, если в них ранее были текст, кроме поля вывода данных о работе симулятора.

Клиент узнаёт о состоянии симулятора с помощью передаваемых ему вместе со страницей флагов, в зависимости от этих состояний кнопки могут становится активными или неактивными, в случае кнопки паузы меняться её отображаемый текст и функционал.

Список входных переменных формируется при помощи JavaScript скрипта, запрашивающего его при старте работы симулятора. Если в ответ не поступило никаких данных, то запрос повторится через секунду, после первого успеха запрос на получение списка входных переменных прекращаются.

Поле выходных данных работы симулятора формируется динамически с помощью ещё одного скрипта. При работе симулятора клиент делает запрос на получение выходных данных для их записи в поле. Если данные пришли, то

повторный запрос отправляется через секунду, если же нет, то клиент начинает отправлять запрос примерно 30 раз в секунду, пока не получит данные. Это сделано для того, чтобы избежать ситуации, когда чтение данных из файла не могло быть произведено сервером, так как в этот момент симулятор записывал туда новые значения.

Верхняя часть интерфейса занимает 65% высоты экрана пользователя. Все элементы являются прокручиваемыми, если данные не помещаются в форму или контейнер.

Пользовательский интерфейс веб-приложения представлен на рисунке 2.

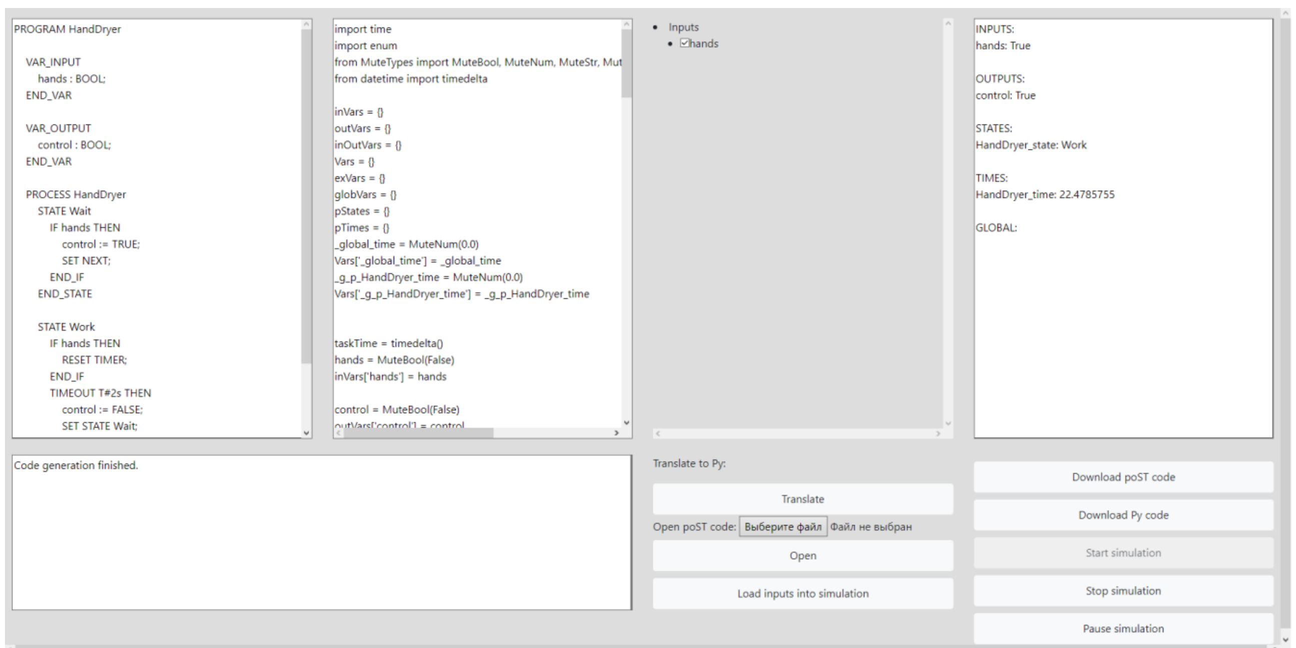

Рисунок 2 — пользовательский интерфейс веб-приложения

3.1.2 Веб-сервер

Серверная часть реализована с помощью веб фреймворка Flask на языке программирования Python. Сервер обрабатывает все запросы, поступающие от клиента, которые он может совершить при помощи пользовательского интерфейса, либо которые совершаются автоматически клиентов автоматически при помощи скриптов на этой странице.

При получении запроса на открытие веб-страницы, сервер проверяет, сохранены ли данные для этого клиента, если да, то он отправляет страницу в том же виде, в котором пользователь видел её в прошлый раз, включая флаги

работы симулятора в случае, если клиент оставил его на исполнение. С одной стороны, это дополнительно нагружает сервер, с другой же, это необходимо, так как подобное длительное использование ПЛК без выключений и перезагрузок является нормальным. Если же у сервера нет сохранённых данных о сессии этого пользователя, то он возвращает страницу с пустыми полями. Данные о сессии пользователя будут сохранены, как только он попытается совершить одно из двух доступных для него в такой момент действия: трансляцию алгоритма на языке poST, написанного в поле ввода, или загрузку файла с готовой программой на сервер, обоих случаях сохраняется файл с программой, в случае трансляции сохраняются также ошибки и предупреждения транслятора и итоговая программа.

При получении запроса на открытие файла с пользовательского компьютера, файл сохраняется в файловой системе сервера и, в составе новой веб-страницы, выводится в изменяемое поле ввода текста.

Команда на трансляцию текст из поля ввода сохраняется в файл, в случае если файл с poST-кодом уже существует, например, в результате загрузки клиентом или в результате прошлой трансляции, он перезаписывается, после чего этот файл подаётся в качестве аргумента на вход исполняемому JAR с транслятором, вывод транслятора сохраняется в отдельный файл и, в случае успеха, создаётся файл с кодом программы на транслированном языке. Текст из файлов отправляется в ответ клиенту в составе заново сформированной веб-страницы в соответствующих текстовых полях, помимо этого, если трансляция прошла успешна и на выходе не пустой файл, то активируется кнопка, с помощью которой можно послать команду на запуск симулятора с получившимся в результате трансляции кодом программы.

В качестве реакции на запрос запуска симулятора сервер копирует все необходимые для этого файлы в директорию сессии пользователя, это необходимо, так как обёрточные классы, используемые сгенерированной программой, находятся в отдельном в отдельном модуле, а значит без него запуск невозможен. После очищаются возможные остаточные данные с

прошлого запуска симуляции, устанавливаются значения флагов запуска симуляции и паузы и записываются в файл для чтения симулятором, также они нужны для формирования изменения функциональности элементов страницы, которая возвращается пользователю.

При вызове остановки симулятора происходят все те же операции, что и при старте, только устанавливаются противоположный запуску флаг запуска симуляции, который во время следующего исполнения этапа чтения цикла сканирования будет считан симулятором и его работа будет завершена. Кроме того, функция остановки симуляции вызывается в случае загрузки нового файла с устройства клиента на сервер и трансляции, на случай если клиент не остановил её самостоятельно.

Если клиент посылает сигнал постановки симулятора на паузу или выхода из неё, то из файла с флагами считывается текущее состояние, значение меняется на противоположное и записывается обратно.

Помимо запросов, которые делаются пользователем, нужно обрабатывать и автоматические запросы, которые делают скрипты для автоматического обновления данных при запуске и работе симулятора. Данные GET-запросы возвращают соответствующий файл со списком входных переменных или выходными данными.

3.2 Тестирование работы виртуального ПЛК на модельной задаче

Для практического тестирования виртуального ПЛК в составе веб-приложения была выбрана задача двух синхронизированных трёхцветных светофоров, одновременно выдающие противоположные сигналы, например, для пешеходов и автомобилей, с одним сенсором, который при активации автоматически включает зелёный цвет на одном из них и красный – на другом. Система имеет 6 выхода и 1 вход.

Сформулированы следующие требования, цвета противоположны для другого светофора:

1. Красный и зелёный сигналы должны гореть по 30 секунд с момента активации, жёлтый – 10 секунд.

2. Жёлтый сигнал всегда должен догореть.
3. При срабатывании сенсора в момент горения зелёного сигнала его продолжительность не должна изменяться.
4. При срабатывании сенсора красный сигнал должен прекратить работу и переключиться на жёлтый.
5. Если на момент окончания работы жёлтого сигнала присутствует сигнал сенсора, то следующим сигналом должен быть зелёный, вне зависимости от того, каким был предыдущий.
6. Для целей внешнего воздействия на состояния и проверку текущего состояния других процессов зажигание сигнала должно быть вынесено в отдельный конфигурируемый процесс.

Для тестирования работы алгоритма использовалось описанное ранее в этой главе веб-приложение, развёрнутом в Docker-контейнере, в который был установлен образ Ubuntu, Python 3.10 и все необходимые модули для работы веб-сервера, виртуальная машина Java и скопированы все основные реализованные средства, необходимые для работы симулятора.

3.2.1 Реализация алгоритма на языке poST

Алгоритм имеет следующую структуру:

8 процессов, реализующие 2 шаблона:

- red_light1 – процесс, отвечающий за зажигание красного сигнала на первом светофоре.
- yellow_light1 – процесс, отвечающий за зажигание жёлтого сигнала на первом светофоре.
- green_light1 – процесс, отвечающий за зажигание зелёного сигнала на первом светофоре.
- red_light2 – процесс, отвечающий за зажигание красного сигнала на втором светофоре.
- yellow_light2 – процесс, отвечающий за зажигание жёлтого сигнала на втором светофоре.

- green_light2 – процесс, отвечающий за зажигание зелёного сигнала на втором светофоре.
- control1 – процесс, контролирующий остальные процессы в первом светофоре.
- control2 – процесс, контролирующий остальные процессы во втором светофоре, синхронизирован по времени с control1 и выдаёт противоположные ему сигналы.

2 шаблона:

- Light – процесс, отвечающий за зажигание сигнала. Имеет один выходной сигнал.
- Control – процесс, контролирующий 3 процесса типа Light. На вход получает массив сигналов, в которых хранятся выходные сигналы процессов Light, этот массив нужен в первую очередь для соблюдения очерёдности исполнения. Для проверки работы обёрточных классов тушение сигналов также происходит через этот массив, в обычных условиях это было бы переключение состояния процессов Light. Имеет 2 входные переменные – массив, про который сказано ранее, и сигнал сенсора, однако массив сам по себе не является входной переменной программы, а потому не должен отображаться в интерфейсе пользователя как поле для ввода.

Синхронизация процесса control2 была произведена просто изменением порядка параметров шаблона, если control1 принимает процессы в порядке: red_light1, yellow_light1, green_light1, то второй в порядке: green_light2, yellow_light2, red_light2. Оба процесса принимают на вход одно значение сенсора.

Полный исходный код алгоритма находится в приложении Б.

3.2.2 Анализ полученных результатов

Разработанный алгоритм на языке poST был транслирован в программу на языке Python с помощью веб-приложения симулятора ПЛК и запущен там же.

Корректность работы проверялась с помощью выводимых данных в текстовое поле для этого предназначенное. Сформулированные требования выполняются.

Кроме того, с помощью этого алгоритма была проверена исправность работы веб-приложения и всех заявленных функций, все встреченные в процессе тестирования недочёты были исправлены.

ЗАКЛЮЧЕНИЕ

В результате работы были проанализированы особенности задач промышленной автоматизации, подходы к организации практических занятий при обучении программированию, отдельное внимание было уделено программированию управляющих систем, подходы к разработке программ для программируемых логических контроллеров, в том числе рассмотрены стандарты IEC 61131-3, IEC 61499 и процесс-ориентированный подход. В результате анализа были сформулированных требования к веб-приложению симулятору ПЛК, с соблюдением которых он был реализован средствами интерпретатора Python.

Для преобразования программы на языке poST в язык интерпретатора Python был реализован транслятор, встраиваемый в ядро языка poST и генерирующий семантически эквивалентный код. Для обхода ограничений, связанных с динамической типизацией языка Python, были реализованы обёрточные классы для неизменяемых типов данных с набором необходимых операторов для работы сгенерированного кода.

Для работы с симулятором было разработано веб-приложение, позволяющее пользоваться транслятором и симулятором удалённо с помощью браузера.

В дальнейшем планируется расширить возможности оператора и добавить возможность соединять несколько симуляторов ПЛК для эмуляции программно-аппаратного моделирования для возможности создания виртуальных лабораторных стендов на их основе и проведения валидации работы управляющих алгоритмов.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

______________________________ ______________________________

ФИО студента *Подпись студента*

« ____ »___________20 __г.

СПИСОК ЛИТЕРАТУРЫ

1. IEC 61131-3-Programmable Controllers–Part 3: Programming Languages // International Electrotechnical Commission. 2013.
2. Зюбин В. Е. LabVIEW: Создание управляющих алгоритмов в процесс-ориентированном стиле // Промышленные АСУ и контроллеры. 2011. № 1. С. 39-45.
3. Зюбин В. Е. Использование виртуальных объектов для обучения программированию информационно-управляющих систем // Информационные технологии, 2009, № 6, С. 79-82.
4. Розов А. С., Зюбин В. Е. Адаптация процесс-ориентированного подхода к разработке встраиваемых микроконтроллерных систем // Автометрия. 2019. Том 55, № 2, С. 114–122.
5. Родченко А. С. Исследование подходов к разработке виртуальных лабораторных стендов в среде CODESYS // МНСК-2019. – 2019. С. 74-74.
6. Bashev V., Anureev I., Zyubin V. The poST language: process-oriented extension for IEC 61131-3 structured text // 2020 International Russian Automation Conference (RusAutoCon). – IEEE, 2020. – С. 994-999.
7. Bashev V., Rozov A., Zyubin V. PoST2ST: A Web Service for Translating poST Programs to the IEC 61131-3 Structured Text // 2021 IEEE 22nd International Conference of Young Professionals in Electron Devices and Materials (EDM), Souzga, the Altai Republic, Russia, 2021, P. 520-523.
8. Christensen J. H., Strasser T., Valentini A., Vyatkin V., and Zoitl A. The IEC 61499 Function Block Standard: Software Tools and Runtime Platforms // ISA Automation Week 2012, 2012.
9. Mader A. A classification of PLC models and applications // Discrete Event Systems: Analysis and Control. – 2000. – С. 239-246.
10. Neamtiu I., Foster J. S., Hicks M. Understanding source code evolution using abstract syntax tree matching // Proceedings of the 2005 international workshop on Mining software repositories. – 2005. – С. 1-5.

11. Werner B. Object-oriented extensions for IEC 61131-3 // IEEE Industrial Electronics Magazine, vol. 3, no. 4, P. 36-39, 2009.
12. Zyubin V. E. Hyper-automaton: A model of control algorithms // 2007 Siberian Conference on Control and Communications. – IEEE, 2007. – С. 51-57.
13. Aho A.V., Ullman J.D. The theory of parsing, translation, and compiling. – Englewood Cliffs, New Jersey: Prentice-Hall, 1972. – Vol. 1. – P. 542. ISBN: 9780139145568.
14. Grinberg M. Flask Web Development: Developing Web Applications with Python. – Sebastopol, California: O'Reilly Media, 2018. P. 312. ISBN: 9781491991732.
15. Robbins J. N. Learning web design: A beginner's guide to HTML, CSS, JavaScript, and web graphics. – Sebastopol, California: O'Reilly Media, 2018. P. 700. ISBN: 9781491960202.
16. Bettini L. Implementing Domain-Specific Languages with Xtext and Xtend. – Birmingham: Packt Publishing, 2016. P. 426. ISBN: 9781786464965.
17. Wagner F. Modeling software with finite state machines: a practical approach. – Boca Raton, Florida: Auerbach Publications, 2006. P. 390. ISBN: 9780849380860.
18. Курс «Программирование ПЛК на основе современного отечественного оборудования FASTWEL I/O» // СПбГЭТУ “ЛЭТИ” [сайт]. URL: https://etu.ru/ru/povyshenie-kvalifikacii/programmy/avtomatizaciya-i-upravlenie/programmirovanie-plk-na-osnove-sovremennogo-otechestvennogo-oborudovaniya-fastwel-io (дата обращения: 16.04.2023).
19. PLC TECHNICIAN TRAINING Online Education Program // PLC Technician [сайт]. URL: https://www.plctechnician.com/ (дата обращения: 16.04.2023)
20. Rust & Docker in production @ Coursera // Medium [сайт]. URL: https://medium.com/coursera-engineering/rust-docker-in-production-coursera-f7841d88e6ed (дата обращения: 16.04.2023).

21. Built-in Types // Python [сайт]. URL: https://docs.python.org/3.10/library/stdtypes.html (дата обращения: 06.04.2023).
22. What’s New In Python 3.10 // Python [сайт]. URL: https://docs.python.org/3.10/whatsnew/3.10.html (дата обращения: 06.04.2023).

ПРИЛОЖЕНИЕ А

Алгоритм исполнения симулятора ПЛК

```
import os
import sys
import time

import poST_code


def main():
    sleepTime = 0
    program = poST_code.Program()
    path = sys.argv[1]
    inputs = "\n".join("{!r}".format(k).replace("\'", "") for k in
poST_code.inVars.keys()) + "\n"
    paused = False
    pauseTime = 0
    pauseStartTime = 0
    with open(path + "/inputs", 'w') as f:
        f.write(inputs)
        f.close()

    while True:
        if not os.path.exists(path + "/flags"):
            with open(path + "/flags", "w") as fl:
                fl.write(False.__str__() + "\n" + False.__str__()
+ "\n")
                fl.close()

        with open(path + "/flags", "r") as f:
            flags = f.read()
            f.close()
        flags = flags.splitlines()
        try:
            stopSim = (flags[0] != "True")
            pauseSim = (flags[1] == "True")
        except:
            pauseSim = True
            stopSim = False

        if pauseSim and not paused:
            paused = True
            pauseStartTime = time.process_time()
        elif not pauseSim and paused:
            paused = False
            pauseFinishTime = time.process_time()
            pauseTime += pauseFinishTime - pauseStartTime
        elif not pauseSim:
            iterStartTime = time.process_time()
            startTime = time.process_time() + sleepTime -
pauseTime
```

```
            poST_code._global_time = startTime

            if not os.path.exists(path + "/sim_in"):
                open(path + "/sim_in", "w").close()
            with open(path + "/sim_in", "r") as f:
                sim_in = f.read()
                f.close()
            sim_in = sim_in.splitlines()
            for key in poST_code.inVars.keys():
                if key in sim_in:
                    try:
                        poST_code.inVars[key].__set__(True)
                    except:
                        pass
                else:
                    try:
                        poST_code.inVars[key].__set__(False)
                    except:
                        pass

            program.run_iter()
            in_text = "INPUTS:\n" + "\n".join(
                "{!r}: {!r}".format(k, v).replace("\'", "") for k,
v in poST_code.inVars.items())
            out_text = "\n\nOUTPUTS:\n" + "\n".join(
                "{!r}: {!r}".format(k, v).replace("\'", "") for k,
v in poST_code.outVars.items())
            states_text = "\n\nSTATES:\n" + "\n".join(
                "{!r}: {!r}".format(k, v).replace("\'", "") for k,
v in poST_code.pStates.items())
            times_text = "\n\nTIMES:\n" + "\n".join(
                "{!r}: {!r}".format(k, v).replace("\'", "") for k,
v in poST_code.pTimes.items())
            glob_text = "\n\nGLOBAL:\n" + "\n".join(
                "{!r}: {!r}".format(k, v).replace("\'", "") for k,
v in poST_code.globVars.items())
            if os.path.exists(path + "/outputs"):
                with open(path + "/outputs", 'r') as f:
                    text = f.read()
                    f.close()
            if text != in_text + out_text + states_text +
times_text + glob_text:
                with open(path + "/outputs", 'w') as f:
                    f.write(in_text + out_text + states_text +
times_text + glob_text)
                    f.close()

            iterFinishTime = time.process_time()
            if poST_code.taskTime is not None:
                if (poST_code.taskTime.total_seconds() > 0):
                    sleepStartTime = time.perf_counter()
                    checkTime = poST_code.taskTime.total_seconds()
```

```
- (iterFinishTime - iterStartTime)
                    if checkTime > 0:
                        time.sleep(checkTime)
                    sleepEndTime = time.perf_counter()
                    sleepTime += (sleepEndTime - sleepStartTime)
            else:
                open(path + "/inputs", 'w').close()
                break
        if stopSim:
            open(path + "/inputs", 'w').close()
            break


if __name__ == '__main__':
    main()
```

ПРИЛОЖЕНИЕ Б

Исходный код модельной задачи на языке poST

```
CONFIGURATION Traffic_lights
    VAR_GLOBAL
       red1 : BOOL;
       yellow1 : BOOL;
       green1 : BOOL;
       red2 : BOOL;
       yellow2 : BOOL;
       green2 : BOOL;
       sensor : BOOL;
    END_VAR

    VAR_GLOBAL CONSTANT
       NUMBER_OF_LIGHTS: INT := 3;
    END_VAR

    VAR_GLOBAL
       lightsArray1 : ARRAY [0 .. NUMBER_OF_LIGHTS] OF BOOL :=
[red1, yellow1, green1];
       lightsArray2 : ARRAY [0 .. NUMBER_OF_LIGHTS] OF BOOL :=
[green2, yellow2, red2];
    END_VAR

    RESOURCE r1 ON Test
       TASK T1 (INTERVAL := T#1s, PRIORITY := 1);
       PROGRAM traffic_lights_controller WITH T1: Controller(
          PROCESS ACTIVE red_light1: Light(b_light => red1),
          PROCESS yellow_light1: Light(b_light => yellow1),
          PROCESS green_light1: Light(b_light => green1),
          PROCESS red_light2 : Light(b_light => red2),
          PROCESS yellow_light2 : Light(b_light => yellow2),
          PROCESS ACTIVE green_light2 : Light(b_light => green2),
          PROCESS ACTIVE control1: Control(control_sensor :=
sensor, pRed := red_light1, pYellow := yellow_light1, pGreen :=
green_light1, rLightsArray := lightsArray1),
          PROCESS ACTIVE control2: Control(control_sensor :=
sensor, pRed := green_light2, pYellow := yellow_light2, pGreen :=
red_light2, rLightsArray := lightsArray2)
       );
    END_RESOURCE
END_CONFIGURATION

PROGRAM Controller
    PROCESS Light
       VAR_OUTPUT
          b_light : BOOL;
       END_VAR

       STATE Light
          b_light := TRUE;
```

```
      END_STATE
   END_PROCESS

   PROCESS Control
      VAR_INPUT
         control_sensor : BOOL;
         rLightsArray : ARRAY [*] OF BOOL;
      END_VAR

      VAR_PROCESS
         pRed : Light;
         pYellow : Light;
         pGreen : Light;
      END_VAR

      VAR
         prev_light : INT;
         alight : INT;
         pressed : BOOL;
      END_VAR

      STATE Work
          IF pressed THEN
            prev_light := 0;
            pressed := FALSE;
         ELSIF ((PROCESS pRed IN STATE INACTIVE) AND (PROCESS
pGreen IN STATE ACTIVE)) OR ((PROCESS pGreen IN STATE INACTIVE)
AND (PROCESS pRed IN STATE ACTIVE)) THEN
            FOR alight := 0 TO NUMBER_OF_LIGHTS DO
               IF rLightsArray[alight] THEN
                  prev_light := alight;
               END_IF
               rLightsArray[alight] := FALSE;
            END_FOR
            STOP PROCESS pRed;
            START PROCESS pYellow;
            STOP PROCESS pGreen;
            SET STATE delay10;
         ELSIF prev_light = 0 THEN
            FOR alight := 0 TO NUMBER_OF_LIGHTS DO
               rLightsArray[alight] := FALSE;
            END_FOR
            STOP PROCESS pRed;
            STOP PROCESS pYellow;
            START PROCESS pGreen;
            SET STATE delay30;
         ELSIF prev_light = 2 THEN
            FOR alight := 0 TO NUMBER_OF_LIGHTS DO
               rLightsArray[alight] := FALSE;
            END_FOR
            START PROCESS pRed;
            STOP PROCESS pYellow;
```

```
            STOP PROCESS pGreen;
            SET STATE delay30;
         END_IF
      END_STATE

      STATE delay10
         TIMEOUT T#10s THEN
            SET STATE Work;
                IF control_sensor THEN
                   pressed := TRUE;
            END_IF
         END_TIMEOUT
      END_STATE

      STATE delay30
         IF control_sensor AND PROCESS pRed IN STATE ACTIVE THEN
             pressed := TRUE;
            SET STATE Work;
         END_IF
         TIMEOUT T#30s THEN
            pressed := FALSE;
            SET STATE Work;
         END_TIMEOUT
      END_STATE
   END_PROCESS
END_PROGRAM
```