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

Мартышкина Дениса Павловича

Тема работы:

Исследование статических методов определения алгоритмической сложности программ на языке poST

«К защите допущена»

Заведующий кафедрой КТ,
д.т.н., доцент
Зюбин В.Е. /……..
(ФИО) / (подпись)
«…» …… 2024г.

Руководитель ВКР

Д.т.н., доцент
зав. каф. КТ ФИТ НГУ
Зюбин В.Е. /……...
(ФИО) / (подпись)
«…» …… 2024г.

Соруководитель ВКР

К.ф-м.н., доцент
каф. КТ ФИТ НГУ
Гаранина Н.О. / ……...
(ФИО)/(подпись)
«…» …… 2024г.

Новосибирск, 2024

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 систем

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)
«…» ..... 20… г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Мартышкину Денису Павловичу

Тема Исследование статических методов определения алгоритмической сложности программ на языке poST.

утверждена распоряжением проректора по учебной работе от 2 ноября № 0409

Срок сдачи студентом готовой работы 20 мая 2024 г.

Исходные данные (или цель работы):

Разработать модуль статического анализа кода на языке poST.

Структурные части работы:

Обзор предметной области, постановка задач, реализация программного обеспечения, апробация на тестовых данных.

Руководитель ВКР
Зав. каф. КТ ФИТ НГУ,
д.т.н., доцент
Зюбин В.Е. / .....
(ФИО) / (подпись)
«…» ..... 20…г.

Задание принял к исполнению
Мартышкин Д.П. / .....
(ФИО студента) / (подпись)
«…» ..... 20…г.

Соруководитель ВКР
Доцент каф. КТ ФИТ НГУ,
к.ф-м.н. Гаранина Н.О. / .....
(ФИО) / (подпись)
«…» ..... 20…г

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В сфере промышленной автоматизации постоянно увеличивается потребность в разработке эффективных методов и средств программного обеспечения для систем управления. Это обусловлено ростом сложности автоматизируемых объектов, расширением их функциональности и увеличением стоимости возможных отказов, особенно вследствие ошибок в программном обеспечении. Повышение качества создаваемых управляющих программ может быть достигнуто с использованием передовых проблемно-ориентированных языков, шаблонов проектирования и методов отладки. Тем не менее, даже на самом совершенном языке программирования можно создать неудачно спроектированную программу, которая будет слабо структурирована, трудна для понимания и сложна в модификации. Ошибки проектирования оказывают крайне негативное влияние на успешность проекта, увеличивая сроки выполнения, превышая запланированные объемы финансирования, что может привести к остановке проекта.

Постоянный контроль качества управляющих программ с самых ранних стадий разработки является необходимым условием для успешного завершения проекта. Это означает, что контроль кода должен осуществляться еще до пуско-наладочных испытаний, когда из-за отсутствия объекта управления или его симулятора невозможно запустить программу. Соответственно, контроль кода должен проводиться методами статического анализа, то есть без запуска программы. Экспертная оценка в виде ревью кода и контроля документации не решает проблему полностью, так как эти методы субъективны и трудоемки. Поэтому разработка автоматических средств контроля параметров исходного кода вызывает интерес не только у теоретиков промышленного программирования, но и у практиков, занимающихся непосредственным созданием управляющих программ.

Для языков общего назначения метрики и алгоритмы оценки алгоритмической сложности достаточно хорошо разработаны. Однако они не

вполне подходят для программ, написанных на специализированных языках, таких как процесс-ориентированные, в частности poST. На данный момент отсутствуют эффективные средства статического анализа для процесс-ориентированных языков программирования, что делает актуальной задачу разработки и внедрения таких инструментов.

Цель работы – разработка методов и средств статического анализа процесс-ориентированных программ на языке poST.

Для достижения цели были определены следующие задачи:

1. Провести анализ существующих метрик для алгоритмической сложности программ, а также особенностей процесс-ориентированных языков и инструментальных средств их поддержки.
2. Сформировать список требований к модулю статического анализа алгоритмической сложности.
3. Определить метрики, адаптированные к процесс-ориентированным программам, и алгоритмы их вычисления.
4. Реализовать алгоритмы определения метрик процесс-ориентированного кода.
5. Исследовать разработанные программные средства на тестовых данных.

Новизна работы заключается в разработке уникального набора метрик кода процесс-ориентированных программ, например, таких, как число процессов, общее число состояний в процессах, число переходов между процессами и другие, на языке poST, который позволяет значительно сократить трудоемкость анализа кода на этом языке, упростить поиск мест в программе, вызывающих сложность в сопровождении и потенциально требующих рефакторинга.

Работа изложена в трех главах. В первой главе приведены результаты анализа существующих подходов и особенностей процесс-ориентированных языков, сформирован список требований. Во второй главе описываются

предлагаемые программные метрики, адаптированные к процесс-ориентированным спецификациям, и алгоритмы их вычисления. Третья глава посвящена вопросам реализации программного модуля и обсуждению результатов его экспериментальной апробации.

Глава 1. Анализ существующих подходов и специфики процесс-ориентированного программирования

1.1. Анализ способов определения алгоритмической сложности программ

Применение программных метрик – это устоявшийся метод позволяющий оценивать различные свойства создаваемого или уже существующего программного обеспечения, прогнозировать объем работ, качества разработанных систем и их частей, характеризовать сложность и надежность программного обеспечения, основываясь на определенных характеристиках кода. Метрика представляет собой функцию, входные значения которой – программный код, выходные – некоторое число. На основе отслеживания определенных метрик проекта можно своевременно обнаружить возникновение нежелательных ситуаций и устранить последствия непродуманных решений.

Для оценки программного кода было предложено множество методов. Среди наиболее часто упоминаемых – lines of code (LoC), цикломатическая сложность Маккейба, программная метрика Холстеда. Метрики можно разбить на три основные группы: количественные, метрики сложности потока управления программ и метрики сложности потока данных программ [1]. Метрики первой группы базируются на определении количественных характеристик, связанных с размером программы, и отличаются относительной простотой. Метрики второй группы базируются на анализе управляющего графа программы. К последней группе относят метрики базирующиеся на оценке использования, конфигурации и размещения данных в программе.

Lines of code (LOC). Значение LOC получается подсчетом числа строк исходного кода, такая метрика проста в расчетах, но не дает точной оценки сложности кода, потому что не учитывает его функциональность [1]. Относится к первой группе.

Показатель сложности Холстеда. Метрикой, учитывающей сложность программных инструкций, являются показатели сложности Холстеда [2, 3], определяющие класс оценок, которые основываются на подсчете числа операторов и операндов. Такой метод дает оценки сложности понимания программы и усилий по её написанию. Также, как и LOC принадлежит к первой группе. Формулы метрики приведены в таблице 1.

Таблица 1 — Формулы показателей Холстеда

Метрика	Формула
Словарь программы	$n = n_1 + n_2$
Длина программы	$N = N_1 + N_2$
Теоретическая длина программы	$N' = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2$
Объем программы	$V = N \cdot \log_2 n$
Трудоемкость кодирования программы	$D = (n_1 \cdot N_2) / (2 \cdot n_2)$
Оценка необходимых интеллектуальных усилий при разработке программы	$E = D \cdot V$

n_1 — число уникальных операторов программы,

n_2 — число уникальных операндов программы,

N_1 — общее число операторов в программе,

N_2 — общее число операндов в программе.

Метрика сложности Маккейба. Для оценивания сложности потока управления программ Маккейбом была предложена метрика, основанная на количестве линейно независимых маршрутов через программный код – цикломатическая сложность, характеризующая трудоемкость тестирования

программы [2, 3]. Программы с более низкой цикломатической сложностью легче понять и менее рискованно модифицировать. К минусам можно отнести нечувствительность к размеру программы, а также нечувствительность к изменению ее структуры.

При вычислении цикломатической сложности используется граф потока управления программы. Узлы графа соответствуют неделимым группам команд программы, а дуги – переходам между ними.

$M = E - N + 2$

M — цикломатическая сложность,

E — количество рёбер в графе,

N — количество узлов в графе.

Метрика спена. Определение спена основывается на локализации обращений к данным внутри каждой программной секции. Спен – это количество утверждений, содержащих данный идентификатор, между его первым и последним появлением в тексте программы. Идентификатор, появившийся n раз, имеет спен, равный n - 1. При большом спене усложняется тестирование и отладка. [1].

Колмогоровская сложность. Колмогоровская сложность представляет собой концепцию из области теоретической информатики, которая измеряет длину кратчайшей программы, способной воспроизвести заданную строку данных. Эта программа считается "минимальной" в смысле того, что нет другой программы, которая была бы короче и производила бы ту же самую строку. Таким образом, колмогоровская сложность оценивает "информативность" строки [4].

Ключевым моментом в понимании колмогоровской сложности является её зависимость от модели вычислений. На практике это означает выбор конкретного языка программирования, на котором программа реализована. От этого выбора зависит абсолютное значение колмогоровской сложности. Однако,

независимо от выбора языка, общий тренд в сложности будет сохраняться, даже если абсолютные значения будут различаться.

Однако у колмогоровской сложности есть одна важная проблема: она является неразрешимой в общем случае. Невозможно создать алгоритм, который для любой строки вычислит ее точную колмогоровскую сложность. Тем не менее, в практических приложениях можно использовать приближенные методы. Сжатие данных может служить индикатором колмогоровской сложности: если фрагмент кода сжимается до определенного размера, это указывает на его верхнюю границу сложности [5]. Для этой задачи подходят многие архиваторы, такие как ZIP, GZIP, RAR и другие.

В контексте анализа кода, колмогоровская сложность может применяться для сравнения одинаковых по функционалу программ как между собой, так и по некоторому эталонному значению.

На основании анализа сделан вывод, что не существует ни одной универсальной метрики, любые метрические характеристики анализируемых программ должны использоваться совместно, кроме того для большинства метрик не существует и эталонных значений, они сравниваются для различных программ и модулей.

1.2. Анализ специфики процесс-ориентированного способа спецификации алгоритмов

Основная идея заключается в применении взаимодействующих процессов, представленных в форме расширенного конечного автомата. Такой метод позволяет более эффективно описывать алгоритмы автоматизации.

Главным понятием процесс-ориентированного подхода является гиперпроцесс — совокупность взаимодействующих процессов, организующих общую память и синхронизацию между собой.

Процесс представляет собой расширенный конечный автомат с набором состояний и переходов. Расширение включает средства синхронизации

выполнения, такие как управление временными интервалами и тайм-аутами. Это достигается путем сохранения времени начала каждого состояния и возможности его сброса.

Состояние включает в себя набор действий, описанных на языке программирования, и состоит из следующих операторов:

- Перевод текущего процесса в другое состояние.
- Запуск, остановка или аварийная остановка текущего или другого процесса.
- Проверка состояния конкретного процесса: активен, неактивен, остановлен или аварийно остановлен.
- Тайм-аут для контроля пребывания процесса в текущем состоянии.

Состояния могут быть обычными или замкнутыми (самопереход). Это разделение не влияет на выполнение программы и необходимо для корректного проектирования алгоритмов. Замкнутые состояния должны быть отмечены как таковые, чтобы избежать ошибок.

В каждом цикле программы процессы выполняют одно из своих состояний и могут быть остановлены или аварийно остановлены. Это достигается добавлением каждому процессу состояний без действий: нормальная остановка (STOP) и аварийная остановка (ERROR). Вывести процесс из этих состояний может только другой процесс.

В начале работы программы все процессы, кроме первого, находятся в состоянии нормальной остановки и запускаются по мере необходимости.

Программа состоит из множества параллельно выполняемых процессов, которые могут запускать, останавливать и контролировать друг друга. Параллелизм в данном контексте является логическим, то есть процессы выполняются последовательно в одном потоке.

1.2.1 Язык poST

Язык Structured Text (ST), являющийся частью стандартов IEC 61131-3, представляет собой мощное средство для реализации систем управления. Однако использование только конструкций общего назначения и отсутствие высокоуровневых абстракций усложняет создание алгоритмов автоматизации, повышая затраты на разработку и усложняя поддержку программ [6].

Процесс-ориентированное программирование, напротив, предлагает более удобный текстовый подход к разработке алгоритмов автоматизации благодаря новому уровню абстракции — процессам.

Язык poST объединяет оба этих инструмента: он является расширением языка ST, что позволяет легко интегрировать его в существующие системы на базе IEC 61131-3, а также является процесс-ориентированным языком, что упрощает разработку и сопровождение управляющих программ.

Таким образом, poST представляет собой процесс-ориентированное расширение языка ST из семейства языков IEC 61131-3. Он предназначен для разработки программного обеспечения для программируемых логических контроллеров и ориентирован на задачи промышленной автоматизации.

Для языка poST разработаны: базовое ядро, генератор кода на языке C, плагин для среды разработки Eclipse IDE и отдельная Java-библиотека (JAR) с транслятором в язык C. Базовое ядро включает в себя парсер, абстрактное синтаксическое дерево (AST), синтаксические и семантические проверки, а также функции, необходимые для IDE, такие как подсветка синтаксиса, выделение ошибок, автодополнение и навигация. Разработка ядра осуществлялась с использованием фреймворка Xtext.

1.2.2. Xtext

Xtext — это программная среда с открытым исходным кодом, предназначенная для создания языков программирования и доменно-специфических языков (DSL). Она использует технологии генерации парсеров,

что позволяет разрабатывать собственные языки программирования через описание грамматики.

С помощью Xtext можно определить собственный язык программирования с использованием специального языка для описания грамматик. На основе этого описания Xtext автоматически генерирует полную инфраструктуру для разработки на целевом языке, включая классы для хранения абстрактного синтаксического дерева (AST), а также готовые парсер и редактор кода [7].

1.2.3. Анализ грамматики языка poST для построения графа потока управления

В контексте статического анализа кода важно обратить внимание на граф потока управления. Этот граф помогает визуализировать порядок выполнения операторов в программе, что значительно упрощает понимание кода, выявление потенциальных проблем и отладку.

Граф потока управления — это ориентированный граф, в котором узлы представляют операции или блоки кода, а ребра указывают на последовательность их выполнения. Построение такого графа требует четкого определения вершин и ребер. Для этого необходимо разобраться в грамматике языка, на котором написан код, так как граф потока управления строится на основе этой грамматики.

Создание программы для автоматического построения графа потока управления начинается с анализа грамматики языка poST. В данном языке особое внимание уделяется концепциям процессов и состояний, что делает разбор грамматики неотъемлемым шагом перед созданием графа. Необходимо определить основные элементы грамматики языка, которые будет использовать создаваемая программа для построения графов. Это включает в себя описание ключевых конструкций языка, таких как процессы и состояния, с учетом их синтаксического описания.

Далее следует описание ключевых элементов грамматики языка и их синтаксиса, представленное в таблице 2. Это позволит создать единые и четкие правила для построения графов потока управления, обеспечивая точность визуализации кода на языке poST.

Таблица 2 — Описание конструкций грамматики poST.

Нетерминальный символ синтаксиса poST	Синтаксис	Описание
Process	**PROCESS** <Имя процесса> <Тело PROCESS> **END_PROCESS**	Описывает процесс, который представляет собой некоторые операции или блоки кода. Тело процесса состоит из набора переменных и состояний.
State	**STATE** <Имя состояния> <Тело STATE> **END_STATE**	Описывает состояние процесса. Состояние может быть замкнутым, что означает, что оно может быть достигнуто только из самого себя. Тело состояния состоит из императивного кода на языке ST и других операторов.
StartProcessStatement	**START PROCESS** <Имя процесса>	Оператор перевода указанного процесса в

		"начальное состояние", то есть запуск процесса.
StartProcessStatement	**RESTART**	Оператор перевода текущего процесса в "начальное состояние", выполняя перезапуск процесса.
StopProcessStatement	**STOP PROCESS** <Имя процесса>	Оператор перевода указанного процесса в выделенное состояние "нормальная остановка", что означает нормальное завершение процесса.
ErrorProcessStatement	**ERROR PROCESS** <Имя процесса>	Оператор перевода указанного процесса в выделенное пассивное состояние «остановка по ошибке» (ошибка процесса).
ErrorProcessStatement	**ERROR**	Оператор перевода текущего процесса в выделенное пассивное состояние «остановка по ошибке» (ошибка процесса).

SetStateStatement	**SET STATE** <Имя состояния>	Оператор перевода текущего процесса в указанное состояние
SetStateStatement	**SET NEXT**	Оператор перевода текущего процесса в следующее по порядку состояние, если оно существует
TimeoutStatement	**TIMEOUT** <Условие> **THEN** <Тело TIMEOUT> **END_TIMEOUT**	Оператор контроля времени нахождения текущего процесса в текущей состоянии. Условием может быть, как временная константа, так и временная переменная.
ResetTimerStatement	**RESET TIMER**	Оператор сброса времени запуска текущего состояния.
AssignmentStatement	<Имя переменной> **:=** <Выражение>	Оператор присваивания значения символьной переменной или переменной массива. Значение для присваивания определяется выражением.

IfStatement	**IF** <Условие> **THEN**
	  <Тело IF>
	ELSIF <Условие>
	THEN
	  <Тело ELSEIF>
	ELSE
	  <Тело ELSE>
	END_IF
		Условный оператор. Условием является «выражение»

CaseStatement	**CASE** <Выражение> **OF** <Список элементов> **ELSE** <Тело ELSE> **END_CASE**	Условный оператор. Выражение представляет проверяемое значение для выбора соответствующего блока кода. Каждый элемент содержит значение или переменную для сравнения и связанный с ним блок кода.
ForStatement	**FOR** <Переменная>**: =** <Значение> **TO** <Значение> **DO** <Тело FOR> **END_FOR**	Оператор цикла для повторения блока кода, заданного в теле, определенное количество раз. Значение переменной меняется в диапазоне от первого значения до второго.

WhileStatement	**WHILE** <Условие> **DO** **END_WHILE**	Оператор цикла для повторения блока кода, заданного в теле, до тех пор, пока выполняется условие.
RepeatStatement	**REPEAT** <Тело REPEAT> **UNTIL** <Условие> **END_REPEAT**	Оператор цикла для повторения блока кода, заданного в теле, до тех пор, пока условие не станет истинным.
FunctionCall	<Имя функции> ‘(‘ <параметры> ‘)’	Оператор вызова функции с переданными в нее аргументами.
SubprogramControlStatement	**RETURN**	Оператор возвращения из функции.
ExitStatement	**EXIT**	Оператор остановки цикла до его завершения.

На основании проведенного анализа грамматики и синтаксиса языка poST были определены правила построения графа потока управления, включая определение вершин и ребер графа:

1. Вершины графа представляют собой различные операторы, такие как Process и State, которые описывают процессы и состояния в рамках программы, условные операторы (IfStatement, CaseStatement), операторы циклов (ForStatement, WhileStatement, RepeatStatement), операторы контроля времени (TimeoutStatement), а также

процессорные операторы (StartProcessStatement, StopProcessStatement, ErrorProcessStatement)

2. Ребра графа устанавливаются между вершинами и указывают на последовательность выполнения операторов. В случае необходимости ребро помечается указанием на грамматическую конструкцию, которую оно представляет.
3. Условные операторы IfStatement и CaseStatement формируют ветвления в графе. Ветвления отображаются ребрами, начинающимися с условия и завершающимися на первом операторе, который выполняется после ветвления.
4. Циклические операторы ForStatement, WhileStatement и RepeatStatement, образуют циклы в графе. Вход и выход из цикла определяются началом и концом соответствующего оператора.
5. Операторы ExitStatement представляют завершение выполнения цикла до его окончания.
6. Процессорные операторы StartProcessStatement, ErrorProcessStatement и StopProcessStatement отображают начало и остановку процессов. Они создают ребра между своими операторами и соответствующими вершинами Process.
7. Операторы SetStateStatement отображают переход текущего процесса в другое состояние, образуя ребро между предыдущим оператором и соответствующей вершиной State. Ребро помечается фразой «SET», если происходит переход в определенное состояние, указанное в операторе, если же указывается переход в следующее состояние, то ребро помечается «SET NEXT».
8. Операторы ResetTimerStatement отображают перезапуск таймера с момента запуска текущего состояния и представлены вершиной.

1.3. Требования к программному модулю

Были сформулированы следующие требования к разрабатываемому модулю:

1. Модуль должен отображать результаты своей работы в формате JSON, как список значений метрик, а также в виде графа потока управления в формате DOT.
2. Модуль должен определять следующие метрики алгоритмической сложности: число строк кода, Холстеда, цикломатическую, спена, колмогоровскую сложность, также метрики специфичные для процесс-ориентированных языков.
3. Метрики должны вычисляться для программы в целом, а также для структурных элементов, характерных для процесс-ориентированных программ: состояний и процессов.
4. Модуль должен предоставлять возможность сохранения результатов своей работы в отдельные файлы.

Глава 2. Программные метрики процесс-ориентированных спецификаций

2.1. Определение цикломатической сложности poST-программ

Для определения цикломатической сложности poST-программ используется граф потока управления, построение которого было рассмотрено ранее. Этот граф позволяет визуализировать последовательность выполнения операторов в программе и является основой для расчета цикломатической сложности.

Для вычисления цикломатической сложности по графу потока управления poST-программы, следует выполнить следующие шаги:

1. Построение графа потока управления: сначала необходимо построить граф потока управления программы, где узлы соответствуют операторам программы, а рёбра указывают на переходы между ними. Особое внимание следует уделить условным и циклическим операторам, так как они создают дополнительные ветвления и циклы в графе.
2. Подсчет узлов: определить количество узлов в графе.
3. Подсчет рёбер: определить количество рёбер в графе.
4. Применение формулы цикломатической сложности: использовать формулу, описанную в первой главе для вычисления метрики, подставив значения, полученные на двух предыдущих шагах.

2.2. Определение колмогоровской сложности poST-программ

Продолжая затронутую тему колмогоровской сложности, данное исследование переходит к применению практического метода для её оценки – сжатия данных. Сжатие представляет собой процесс оптимизации хранения информации путем ее уплотнения. Такой подход позволяет получить

приближенную оценку колмогоровской сложности, которая в прямом вычислении является неразрешимой задачей.

Однако, при использовании различных методов сжатия, как например форматы ARJ, RAR и ZIP, необходимо учитывать, что помимо самой сжатой информации, создается и служебная информация. Служебная информация играет важную роль в восстановлении сжатых данных и ее наличие является обязательной частью архивных файлов. Размер этой дополнительной информации зависит от используемого алгоритма и может быть значимым. Поэтому для корректности исследования и исключения "перекоса" в колмогоровскую сложность за счет служебной информации, следует оценить объем служебной информации, архивируя пустой файл с расширением .post для каждого из исследуемых форматов. Размер получившегося архива представляет минимальный объем служебной информации, который создаёт конкретный алгоритм сжатия.

Ручной разбор архива для извлечения только сжатых данных – процесс весьма трудоемкий. Это связано с необходимостью детального анализа структуры файла, разбора заголовков, маркеров и других служебных элементов, которые определяют организацию и формат данных внутри архива. Такой процесс требует глубокого понимания используемых алгоритмов сжатия данных и архивных форматов, и может потребовать значительных усилий и времени. Поэтому для приближенной оценки размера сжатых данных предлагается использовать перечисленный подход, при котором из общего объема архива вычитается минимальный размер служебных данных.

Таблица 3 представляет обзор размера получившихся архивов для файлов .post, в которых содержатся известные программы на данном языке различных размеров. Эти данные помогут получить более точную оценку колмогоровской сложности исследуемых программ и упростить процесс оценки, минуя трудоемкий ручной разбор архива.

Таблица 3 – полученные размеры архивов

	Исходный размер, байт	RAR, байт	ZIP, байт	GZIP, байт
Пустой файл	0	68	153	97
Dryer	750	389	446	439
Elevator	12076	2032	2142	2042
CUKM	1 163 188	74703	94425	71383

Размер архива также зависит и от длины названий, входящих в него файлов, каждый символ имени кодируется одним байтом. Для данных в таблице уже учтено кодирование имен, то есть их длина считается нулевой.

Для вычисления колмогоровской сложности предлагается использовать такую формулу: KC = AS - (EFAS + FNL), где

KC - Kolmogorov complexity,

AS - archive size,

EFAS - empty file archive size,

FNL - file name length.

2.3. Специфичные метрики для процесс-ориентированных языков

Были предложены следующие метрики:

1. Число процессов: большое количество процессов может указывать на сложность системы или обширный набор бизнес-процессов, которые должны быть учтены. Позволяет оценить сложность программы.
2. Общее число состояний процессов: большое количество состояний может указывать на более сложные процессы или наличие разнообразных вариантов состояний, которые требуют анализа.
3. Общее число инструкций в состояниях процессов: большое количество инструкций может указывать на детализированные или сложные действия, которые должны быть выполнены в каждом состоянии.

Позволяет оценить объем работы, требуемой для реализации и управления процессами.

4. Среднее число состояний: высокое среднее число состояний в процессе может указывать на более детализированные процессы. Может быть полезна для определения типичного размера и сложности процессов.
5. Среднее число инструкций: высокое среднее число инструкций может указывать на более сложные или времязатратные процессы. Позволяет оценивать объем работы, требуемый для реализации и выполнения каждого процесса.
6. Число переходов между процессами: высокое число переходов может указывать на интенсивное взаимодействие и коммуникацию между процессами. Может быть полезно для оптимизации программы, так как позволяет оценить не только отдельные процессы, но и их взаимодействие, кроме того дает представление о сложности и объеме работы, связанных с моделью процессов.

Эти метрики могут помочь разработчикам и аналитикам получить представление о сложности и размере моделей процессов, а также помочь в оценке ресурсозатратности, анализе и планировании разработки и оптимизации процессов.

Глава 3. Реализация модуля вычисления программных метрик и результаты практической апробации

Для реализации программного модуля использовался Java-подобный язык Xtend с редким применением Java.

3.1. Особенности языка Xtend

Xtend – это статически типизированный язык программирования, транслирующийся в Java [8]. Синтаксически и семантически Xtend имеет корни в языке программирования Java, с которым имеет полную совместимость, но обладает некоторыми особенностями. Основные из них:

1. Вывод типов

   В Xtend редко требуется явно указывать типы переменных или выражений благодаря механизму вывода типов. Компилятор Xtend способен вывести тип переменной или выражения из контекста, где оно используется. Это уменьшает необходимость повторного написания типов и делает код более компактным и легким для чтения. Для того чтобы задать функцию или неизменяемое выражения используются ключевые слова def и val соответственно.

2. Методы расширения

   Позволяют добавлять новые методы к существующим типам данных, не модифицируя эти типы напрямую. Они предоставляют возможность вызывать эти новые методы на объектах так, как будто они являются частью исходных типов.

3. Умное приведение типов

   Вместо того чтобы делать приведение типов явно, как это часто приходится делать в Java, Xtend автоматически приводит объект к проверяемому типу в операторе instanceof. Это означает, что, если объект является экземпляром или наследником указанного типа, то

возможно работать с объектом, как с этим типом, без необходимости явного приведения. Такая возможность очень полезна при частой проверке на принадлежность к тому или иному классу.

4. Шаблонные выражения

   Предоставляют удобный способ объединения строк. Шаблоны заключаются в тройные одинарные кавычки (''). Шаблон может занимать несколько строк, а также содержать вложенные выражения, заключенные в специальные кавычки. Внутри них можно использовать условия и циклы.

3.2. Структура модуля и его реализация

Модуль разбит на несколько пакетов, состоящих из Java и Xtend классов. Рассмотрим их детальнее.

3.2.1. Пакет api

Представляет собой набор абстрактных классов и интерфейсов, которые используются для разделения функциональности на уровне приложения. Они позволяют избежать прямой зависимости от конкретных реализаций классов и обеспечивают более гибкую архитектуру.

IArchiver – интерфейс для различных архиваторов. Содержит один метод create, который предназначается для создания архива конкретного вида и должен быть переопределен в классах-наследниках.

BaseArchiver – абстрактный базовый класс для архиваторов, реализующий IArchiver. Он предоставляет логику для инициализации путей и имен: имени архива, имени исходного файла для архивации, а также путей их расположения в файловой системе. Не реализует метод create родительского интерфейса, оставляя это для более специфичного класса-наследника.

ICmdRunner – интерфейс, используемый для архиваторов, которые создают свои архивы с помощью системных команд. Содержит метод formAndRunCommand отвечающий за формирование и запуск команды.

IMetric – интерфейс от которого наследуются все классы метрик. Не содержит атрибутов и методов.

IMetricCalculator – интерфейс для различных видов калькуляторов метрик. Содержит один метод calculate, возвращающий объект типа IMetric, в котором содержится рассчитанное значение, реализация предоставляется классам-наследникам.

3.2.2. Пакет config

Этот пакет содержит конфигурационные файлы и классы для управления конфигурацией всего приложения. Он отвечает за централизованное хранение настроек, параметров и конфигураций, используемых вашим приложением.

PathConfig – класс, содержащий static поля, в которых хранятся расположения для директории post проекта, директорий src и src-gen, а также текстовое поле для расширения post программ “.post”. Используется в тех местах где нужен доступ к этим директориям, например, в BaseArchiver.

3.2.3. Пакет dto

Состоит из классов данных, которые представляют собой простые объекты для передачи данных между различными частями приложения. Они предназначены для удобства передачи данных.

IfElseBlock – класс данных, который используется для соединения вершин графа потока управления, связанными с условными операторами, ребрами. Содержит два атрибута: ссылку на вершину графа, из которой должно идти ребро в вершину, отвечающей за выражение, идущее после условного оператора, а также булеву переменную, значение которой ложно, если в

IfElseBlock хранится ссылка на вершину условного оператора и истинно, если ссылка указывает на вершину из блока else.

3.2.4. Пакет generator

Является основным пакетом программы. Содержит Main класс, код которого представлен в приложении А.

Main – главный класс программы. Реализует интерфейс IPostGenerator, который наследует интерфейс IPostGenerator2 фреймворка Xtext [9]. Содержит три метода: beforeGenerate, doGenerate, afterGenerate. В afterGenerate освобождаются и сбрасываются данные, которые больше не нужны. В beforeGenerate происходит инициализация всех необходимых полей. В doGenerate расположена логика по построению графа потока управления, расчету метрик, сохранению всех результатов в отдельные файлы.

3.2.5. Пакет metric

Представлен классами данных, предназначенные для хранения метрик кода. Эти классы представляют собой контейнеры для значений метрик, которые считаются в классах-калькуляторах, а затем сериализуются в формате json. Все классы в этом пакете наследуют интерфейс IMetric, а также помечены аннотациями Xtend [10] для автоматической генерации геттеров и сеттеров их полей, использование которых позволяет значительно сократить шаблонный код.

CyclomaticMetric – класс данных для значения цикломатической сложности Маккейба.

HalsteadMetric – класс данных для значений метрик Холстеда.

LinesOfCodeMetric – класс данных для значения метрики числа строк кода.

MetricsAggregate – класс данных, который агрегирует значения всех метрик. Используется для дальнейшей сериализации.

PostSpecificMetric – класс данных для значений специфичных метрик процесс-ориентированных языков.

3.2.6. Пакет util

Содержит различные инструментальные классы и утилиты, которые могут быть легко использованы в различных частях программы, что позволяет избежать дублируемости кода. Этот пакет предоставляет удобный способ организации и доступа к различным вспомогательным функциям. Состоит из четырех входящих в него пакетов.

3.2.7. Пакет util.metric

Пакет для хранения классов-калькуляторов значений метрик, представленных в пакете metric. Предоставляет удобный способ организации и доступа к различным калькуляторам, которые используются для и оценки качества кода. Все классы отсюда наследуются интерфейс IMetricCalculator и реализуются его единственный метод calculate.

PostSpecificMetricCalculator – калькулятор, отвечающий за расчет значений метрик специфичных для процесс-ориентированных программ. Метод calculate возвращает объект типа PostSpecificMetric. Значения считаются на основании AST дерева программы.

LinesOfCodeMetricCalculator – калькулятор, отвечающий за значение числа строк кода. Метод calculate возвращает объект типа LinesOfCodeMetric. Значение рассчитывается основываясь на файле исходной программы с расширением .post.

HalsteadMetricCalculator – калькулятор нужный для нахождения значений метрик Холстеда. Метод calculate возвращает объект типа HalsteadMetric.

CyclomaticMetricCalculator – калькулятор цикломатической сложности Маккейба. Метод calculate возвращает объект типа CyclomaticMetric.

Значение считается на основании графа потока управления по формуле, приведенной в разделе, которому посвящена эта метрика.

3.2.8. Пакет util.dot

Отвечает за набор инструментальных классов необходимых для работы с графами в формате DOT, который используется для их описания и визуализации.

DotGenerator – класс представлен методом generateRepresentatiot, который принимает ссылку на объект графа потока управления. Метод проходит по ребрам и вершинам полученного графа, на выход подается текст - строковое представления графа. Для формирования строки используется StringBuilder, позволяющий оптимизировать её частое изменение.

3.2.9. Пакет util.converter

Содержит конвертеры, предназначенных для преобразования данных из одного формата в другой.

MetricsJsonConverter – конвертер, обеспечивающий преобразования метрик в формат Json. Он содержит перегруженные методы toJson для конвертации списка метрик или отдельной метрики в JSON-строку, используя библиотеку Gson для сериализации объектов. Метод принимающий список проходит по нему, для каждой метрики вызывает второй метод, а также добавляет к результирующей строке заголовки соответствующих метрик из списка.

3.2.10. Пакет util.archive

Представляет собой набор инструментов, специализированных для работы с архивами. Он содержит различные классы, включая различные виды архиваторов, фабрику для создания архиваторов, класс-счетчик размера архива и перечисление, определяющее типы архивов.

ArchiveAnalyzer – инструмент для анализа архива. Включает один метод getArchiveSize, принимающий один строковый параметр - путь до архива и возвращающий размер этого архива в байтах.

ArchiveFormat – java перечисление, определяющие поддерживаемые типы архивов. Каждая константа перечисления представляет собой определенный формат архива и связанный с ним архиватор, который реализует интерфейс IArchiver. Этот класс предоставляет удобный способ создания экземпляров архиваторов для конкретных форматов архивов в фабрике. В каждой константе перечисления определены как поле, так и метод. Строковое поле extension хранит расширение файла для соответствующего формата архива, а метод getArchiverInstance возвращает экземпляр соответствующего архиватора. Каждая константа перечисления переопределяет этот метод, чтобы вернуть экземпляр связанного с ней архиватора. Выбор java-перечисления связан с тем, что язык Xtend не поддерживает переопределения абстрактных методов перечисления внутри каждой константы. Такая возможность необходима для удобного и безопасного введения новых форматов или удаления старых.

ArchiverFactory – статическая фабрика, которая предоставляет способ получения экземпляров архиваторов для различных форматов архивов. Класс реализован как final, что предотвращает его наследование, и имеет приватный конструктор, чтобы запретить создание экземпляров этого класса извне. Метод getArchiver принимает параметр типа ArchiveFormat, который определяет требуемый формат архива, и возвращает соответствующий архиватор с помощью вызова метода getArchiverInstance из перечисления ArchiveFormat. Ответственность за создание экземпляра лежит на конкретной константе перечисления, что позволяет избежать цепочки swtich проверок на конкретный тип формата, которая может приводить к ошибкам при расширении кода.

GZIPArchiver – класс-архиватор для формата GZIP. Наследуется от класса BaseArchiver и реализует интерфейс ICmdRunner. Переопределенный

метод create отвечает за создание GZIP-архива путем выполнения команды в отдельном процессе с помощью сторонней программы 7-Zip. После завершения операции архивации метод возвращает путь к созданному архиву.

RarArchiver – класс-архиватор аналогичный архиватору GZIPArchiver, но отвечающий за формат Rar.

ZipArchiver – класс-архиватор для формата ZIP. Он наследуется от абстрактного класса BaseArchiver и переопределяет метод create, который отвечает за создание архива. Метод create принимает название программы и название проекта, и возвращает строковый путь к созданному архиву. Внутри метода происходит инициализация путей, чтение файлов, создание архива и запись данных в него. Использует стандартные java потоки ZipOutputStream, FileOutputStream и FileInputStream для записи в zip-файл, работа с ними происходит в try-catch-finally блоках для обработки исключительных ситуаций и гарантированного закрытия потоков.

3.2.11. Пакет cfg

Содержит классы и инструменты, необходимые для работы с графом управления потоком. В нем хранятся данные о вершинах и ребрах графа, которые представлены в виде классов Node и Edge. Также в пакете реализованы классы, которые обеспечивают функциональность по его созданию. Работа с графом возможна на уровне программы, процесса или состояния.

Node – класс, представляющий узел графа. Узел имеет уникальный идентификатор, текстовую метку, указатель на объект модели EObject и тип EClass. Класс также содержит методы для создания узла с указанием метки и объекта модели, а также методы для сброса и инкремента текущего идентификатора узла.

Edge – класс, описывающий ребро графа. Содержит ссылки на начальный и конечный узлы ребра, а также текстовую метку, описывающую связь между узлами.

Cfg – класс, представляющий граф управления потоком. Поля представлены списками узлов и ребер. Конструкторы перегружены для нескольких типов уровней программ на языке poST: программы, процесса, состояния, и создают узлы и ребра графа на основе этих объектов с помощью NodeBuilder и EdgeBuilder. Полученные элементы используются для дальнейшего анализа и обработки.

NodeBuilder – класс, предназначенный для работы с узлами графа. Он содержит статические методы для создания узлов на основе типа уровня программы. Каждый метод принимает соответствующий объект и создает узлы графа на основе его структуры и содержимого, спускаясь ниже по уровням, если это возможно, и обрабатывая языковые инструкции.

EdgeBuilder – класс предназначен для построения ребер графа управления на основе полученных инструкций из объектов программы. Он содержит методы для создания ребер графа на основе различных типов утверждений, таких как присваивания, условия, циклы и т. д. Каждый метод выполняет обход по переданным утверждениям и создает соответствующие ребра между узлами графа. Класс также поддерживает создание ребер для различных уровней вложенности, например, в случае вложенных условий или циклов.

3.3. Апробация на тестовом наборе poST-программ

Тестирования модуля статического анализа проводилось путем передачи в редактор кода тестовых программ на языке poST. Одна из них содержит основные конструкции языка, две остальные представляют собой небольшие реальные программы. Их исходный текст представлен в приложении Б. Результаты, полученные модулем, представлены в приложении С.

Процесс проверки корректности состоял в ручном подсчете метрик для каждой программы. Затем использовался модуль анализа на тех же самых программах, чтобы автоматически получить результаты. После этого сравнивались результаты ручного анализа с автоматическими результатами, которые полностью совпали, что подтверждает точность работы модуля.

ЗАКЛЮЧЕНИЕ

В результате выполнения дипломной работы был предложен список метрик специфичных для описанного подхода, а также реализован модуль статического анализа, который позволяет визуализировать граф потока управления и вести расчет значений группы метрик.

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”.

Мартышкин Денис Павлович
ФИО студента

_____
Подпись студента

«____» _____20__.
(Заполняется от руки)

Список использованных источников и литературы

1. Новичков А., Шамрай А., Черников А. Метрики кода и их практическая реализация в Subversion и ClearCase //CM Consulting.-2008. – 2008.
2. Khan A. A. et al. Comparison of software complexity metrics //International Journal of Computing and Network Technology. – 2016. – Т. 4. – №. 01.
3. Gsellmann P., Melik-Merkumians M., Schitter G. Comparison of code measures of IEC 61131–3 and 61499 standards for typical automation applications //2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). – IEEE, 2018. – Т. 1. – С. 1047-1050.
4. Shen A., Uspensky V. A., Vereshchagin N. Kolmogorov complexity and algorithmic randomness. – American Mathematical Society, 2022. – Т. 220.
5. Zenil H., Marshall J. A. R., Tegnér J. Approximations of algorithmic and structural complexity validate cognitive-behavioural experimental results //arXiv preprint arXiv:1509.06338. – 2015.
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. L. Bettini. Implementing Domain-Specific Languages with Xtext and Xtend // Packt Publishing, 2016.
8. Xtend – Introduction [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/xtend/documentation/index.html (дата обращения 12.05.2024)
9. Xtext – IGenerator2 [Электронный ресурс] / – Режим доступа: https://archive.eclipse.org/modeling/tmf/xtext/javadoc/2.9/org/eclipse/xtext/generator/IGenerator2.html (дата обращения 12.05.2024)
10. Xtend – Active Annotations [Электронный ресурс] / – Режим доступа: https://eclipse.dev/Xtext/xtend/documentation/204_activeannotations.html (дата обращения 12.05.2024)

Приложение А.

```
class Main implements IPoSTGenerator{

	protected Model model
	protected Program program

	protected String projectName

	protected String fileName
	protected String cfgFile =  '''processCFG.dot'''

	protected Cfg cfg

	protected IArchiver archiver =  ArchiverFactory.getArchiver(ArchiveFormat.GZIP)


	override setModel(Model model) {
		this.model = model
		this.program = model.programs.get(0)
	}

	override afterGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context)
{
		Node.resetCurrentId()
	}

	override beforeGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext
context) {
		setModel(resource.allContents.toIterable.filter(Model).get(0))

		fileName = '''«program.getName()»_metrics.json'''

		projectName = getPostProjectName(resource.getURI.toPlatformString(false))
		if(projectName === null) {
			throw new NullPointerException("Nullable poST project")
		}
	}

	override doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
		cfg = new Cfg(program)
		fsa.generateFile(cfgFile, DotGenerator.generateRepresentation(cfg))

		var List<IMetric> metrics = CollectionLiterals.newArrayList()
		var cyclomaticMetricCalculator = new CyclomaticMetricCalculator(cfg)
		var postSpecificMetricCalculator = new PostSpecificMetricCalculator(program)
		var linesOfCodeMetricCalculator = new LinesOfCodeMetricCalculator(program.getName,
projectName)
		metrics.add(linesOfCodeMetricCalculator.calculate)
		metrics.add(cyclomaticMetricCalculator.calculate)
		metrics.add(postSpecificMetricCalculator.calculate)

		fsa.generateFile(fileName, MetricsJsonConverter.toJSON(metrics))

		val zipPath = archiver.create(program.getName, projectName)
	}

	private def getPostProjectName(String platformPath) {
		val pattern = Pattern.compile("/(.*?)/");
		val matcher = pattern.matcher(platformPath);
		if (matcher.find()) {
		return matcher.group(1);
		}
		return null;
	}
}
```

Приложение Б.

Листинг тестовых программ.

Программа DemoProject:

```
CONFIGURATION Conf
   RESOURCE Res1 ON TestCPU
       TASK T1 (INTERVAL := T#100ms, PRIORITY := 1);
       PROGRAM demo WITH T1 : DemoProgram;
   END_RESOURCE
END_CONFIGURATION
PROGRAM DemoProgram
   VAR_INPUT
       input1 : BOOL;
       input2 : INT;
   END_VAR
   VAR_OUTPUT
       output1 : BOOL;
       output2 : INT;
   END_VAR
   PROCESS MainProcess
       STATE Init
           IF input1 THEN
               SET NEXT;
           END_IF
       END_STATE
       STATE Execute
           output1 := TRUE;
           output2 := input2 + 1;
           IF input2 > 10 THEN
               output2 := 0;
           ELSE
               output2 := -1;
           END_IF
           FOR output2 := 1 TO 3 DO
               output2 := output2 + 1;
           END_FOR
           WHILE input2 < 10 DO
               output2 := output2 + 1;
           END_WHILE
           REPEAT
               output2 := output2 - 1;
           UNTIL output2 = 0 END_REPEAT
           IF output1 THEN
               RETURN;
           END_IF
           EXIT;
           RESET TIMER;
           SET STATE Idle;
       END_STATE
       STATE Idle
           TIMEOUT T#1s THEN
               SET STATE Init;
           END_TIMEOUT
       END_STATE
   END_PROCESS

   PROCESS SecondaryProcess
       STATE Start
           IF input2 > 5 THEN
               START PROCESS MainProcess;
```

```
            END_IF

            SET NEXT;
        END_STATE
        STATE Stop
            IF input2 < 5 THEN
                STOP PROCESS MainProcess;
            ELSE
                ERROR PROCESS MainProcess;
            END_IF
        END_STATE
    END_PROCESS
END_PROGRAM
```

Программа Elevator:

```
PROGRAM Controller
VAR_INPUT
(* floor sensors *)
onfloor0 : BOOL;
onfloor1 : BOOL;
onfloor2 : BOOL;
(* floor call buttons *)
call0 : BOOL;
call1 : BOOL;
call2 : BOOL;
(* inside car call buttons *)
button0 : BOOL;
button1 : BOOL;
button2 : BOOL;
(* doors sensors *)
door0closed : BOOL;
door1closed : BOOL;
door2closed : BOOL;
END_VAR
VAR_OUTPUT
(* elevator *)
up : BOOL;
down : BOOL;
(* doors *)
open0 : BOOL;
open1 : BOOL;
open2 : BOOL;
(* LEDs for inside/outside call buttons *)
call0_LED : BOOL;
call1_LED : BOOL;
call2_LED : BOOL;
button0_LED : BOOL;
button1_LED : BOOL;
button2_LED : BOOL;
(* cur floor LEDs *)
floor0_LED : BOOL;
floor1_LED : BOOL;
floor2_LED : BOOL;
(* cur floor num *)
cur : INT;
END_VAR
VAR
(* target floor num *)
target : INT;
END_VAR
PROCESS Init (* initial process *)
STATE begin
```

```
        START PROCESS Call0Latch;
        START PROCESS Call1Latch;
        START PROCESS Call2Latch;
        START PROCESS Button0Latch;
        START PROCESS Button1Latch;
        START PROCESS Button2Latch;
        START PROCESS CheckCurFloor;
        START PROCESS UpControl;
        STOP;
END_STATE
END_PROCESS
PROCESS Call0Latch
VAR
        prev_in : BOOL;
        prev_out : BOOL;
END_VAR
STATE init
        prev_in := NOT call0;
        prev_out := NOT open0;
        SET NEXT;
END_STATE
STATE check_ON_OFF LOOPED
        IF call0 AND NOT prev_in THEN
                call0_LED := TRUE;
        END_IF
        IF open0 AND NOT prev_out THEN
                call0_LED := FALSE;
        END_IF
        prev_in := call0;
        prev_out := open0;
END_STATE
END_PROCESS
PROCESS Call1Latch
VAR
        prev_in : BOOL;
        prev_out : BOOL;
END_VAR
STATE init
        prev_in := NOT call1;
        prev_out := NOT open1;
        SET NEXT;
END_STATE
STATE check_ON_OFF LOOPED
        IF call1 AND NOT prev_in THEN
                call1_LED := TRUE;
        END_IF
        IF open1 AND NOT prev_out THEN
                call1_LED := FALSE;
        END_IF
        prev_in := call1;
        prev_out := open1;
END_STATE
END_PROCESS
PROCESS Call2Latch
VAR
        prev_in : BOOL;
        prev_out : BOOL;
END_VAR
STATE init
        prev_in := NOT call2;
        prev_out := NOT open2;
        SET NEXT;
END_STATE
STATE check_ON_OFF LOOPED
        IF call2 AND NOT prev_in THEN
                call2_LED := TRUE;
        END_IF
        IF open2 AND NOT prev_out THEN
```

```
                call2_LED := FALSE;
        END_IF
        prev_in := call2;
        prev_out := open2;
END_STATE
END_PROCESS
PROCESS Button0Latch
VAR
        prev_in : BOOL;
        prev_out : BOOL;
END_VAR
STATE init
        prev_in := NOT button0;
        prev_out := NOT open0;
        SET NEXT;
END_STATE
STATE check_ON_OFF LOOPED
        IF button0 AND NOT prev_in THEN
                button0_LED := TRUE;
        END_IF
        IF open0 AND NOT prev_out THEN
                button0_LED := FALSE;
        END_IF
        prev_in := button0;
        prev_out := open0;
END_STATE
END_PROCESS
PROCESS Button1Latch
VAR
        prev_in : BOOL;
        prev_out : BOOL;
END_VAR
STATE init
        prev_in := NOT button1;
        prev_out := NOT open1;
        SET NEXT;
END_STATE
STATE check_ON_OFF LOOPED
        IF button1 AND NOT prev_in THEN
                button1_LED := TRUE;
        END_IF
        IF open1 AND NOT prev_out THEN
                button1_LED := FALSE;
        END_IF
        prev_in := button1;
        prev_out := open1;
END_STATE
END_PROCESS
PROCESS Button2Latch
VAR
        prev_in : BOOL;
        prev_out : BOOL;
END_VAR
STATE init
        prev_in := NOT button2;
        prev_out := NOT open2;
        SET NEXT;
END_STATE
STATE check_ON_OFF LOOPED
        IF button2 AND NOT prev_in THEN
                button2_LED := TRUE;
        END_IF
        IF open2 AND NOT prev_out THEN
                button2_LED := FALSE;
        END_IF
        prev_in := button2;
        prev_out := open2;
END_STATE
```

```
END_PROCESS
PROCESS CheckCurFloor
STATE check_floor
        IF onfloor0 THEN
                cur := 0;
                floor0_LED := TRUE;
                floor1_LED := FALSE;
                floor2_LED := FALSE;
        ELSIF onfloor1 THEN
                cur := 1;
                floor0_LED := FALSE;
                floor1_LED := TRUE;
                floor2_LED := FALSE;
        ELSIF onfloor2 THEN
                cur := 2;
                floor0_LED := FALSE;
                floor1_LED := FALSE;
                floor2_LED := TRUE;
        END_IF
END_STATE
END_PROCESS
PROCESS UpControl (* up motion priority *)
STATE check_calls
        (* a call from the current floor? *)
                IF      (cur = 0 AND (call0_LED OR button0_LED)) OR
                (cur = 1 AND (call1_LED OR button1_LED)) OR
                (cur = 2 AND (call2_LED OR button2_LED)) THEN
                START PROCESS DoorCycle;
                SET STATE door_cycle;
        ELSE
                (* are there other calls? *)
                CASE (cur) OF (* a call from an upper floor? *)
                        0:      IF ((call1_LED OR button1_LED) OR
                                        (call2_LED OR button2_LED))THEN
                                        START PROCESS UpMotion;
                                        SET NEXT;
                                END_IF
                        1:      IF (call2_LED OR button2_LED) THEN (* above? *)
                                        START PROCESS UpMotion;
                                        SET NEXT;
                                ELSIF (call0_LED OR button0_LED) THEN (* below? *)
                                        START PROCESS DownControl;
                                        STOP;
                                END_IF
                        2:      START PROCESS DownControl; (* switch direction *)
                                STOP;
                        END_CASE
        END_IF
END_STATE
STATE check_stop
        IF (PROCESS UpMotion IN STATE INACTIVE) THEN
                START PROCESS DoorCycle;
                SET NEXT;
        END_IF
END_STATE
STATE door_cycle
        IF (PROCESS DoorCycle IN STATE INACTIVE) THEN
                RESTART; (* set the initial state *)
        END_IF
END_STATE
END_PROCESS
PROCESS UpMotion
STATE start (* check the next floor call *)
        up := TRUE;
        CASE (cur) OF
                0:      IF (call1_LED OR button1_LED) THEN
                                target := 1; (* if next floor *)
                                SET NEXT;
```

```
                        END_IF
                1:
                        IF (call2_LED OR button2_LED) THEN
                                target := 2; (* if next floor *)
                                SET NEXT;
                        END_IF
                2:
                        target := 2; (* a stub *)
                        SET NEXT;
                END_CASE
END_STATE
STATE check_target
        IF (cur = target) THEN  (* have arrived? *)
                up := FALSE;
                STOP;
        END_IF
END_STATE
END_PROCESS
PROCESS DownControl
STATE check_calls
        (* a call from the current floor? *)
                IF      (cur = 0 AND (call0 OR button0)) OR
                (cur = 1 AND (call1 OR button1)) OR
                (cur = 2 AND (call2 OR button2)) THEN
                START PROCESS DoorCycle;
                SET STATE door_cycle;
        ELSE
                (* are there other calls? down or switch *)
                CASE (cur) OF
                        0:      START PROCESS UpControl;
                                STOP;
                        1:      IF (call0_LED OR button0_LED) THEN (* down? go *)
                                        START PROCESS DownMotion;
                                        SET NEXT;
                                ELSIF (call2_LED OR button2_LED) THEN (* up? switch *)
                                        START PROCESS UpControl;
                                        STOP;
                                END_IF
                        2:      IF      ((call1_LED OR button1_LED) OR
                                                (call0_LED OR button0_LED))THEN
                                        START PROCESS DownMotion;
                                        SET NEXT;
                                END_IF
                        END_CASE
        END_IF
END_STATE
STATE check_stop
        IF (PROCESS DownMotion IN STATE INACTIVE) THEN
                START PROCESS DoorCycle;
                SET NEXT;
        END_IF
END_STATE
STATE door_cycle
        IF (PROCESS DoorCycle IN STATE INACTIVE) THEN
                RESTART;
        END_IF
END_STATE
END_PROCESS
PROCESS DownMotion
STATE start
        down := TRUE;
        CASE (cur) OF
                0: (* a stub *)
                        target := 0;
                SET NEXT;
                1:
                        IF (call0_LED OR button0_LED) THEN
                                target := 0; (* if next floor *)
```

```
                        SET NEXT;
                END_IF
        2:
                IF (call1_LED OR button1_LED) THEN
                        target := 1; (* if next floor *)
                        SET NEXT;
                END_IF
        END_CASE
END_STATE
STATE chech_next
        IF (cur = target) THEN (* have arrived? *)
                down := FALSE;
                STOP;
        END_IF
END_STATE
END_PROCESS
PROCESS DoorCycle
STATE choose_door_to_open
        CASE (cur) OF
                0:  open0 := TRUE;
                1:  open1 := TRUE;
                2:  open2 := TRUE;
                END_CASE
        SET NEXT;
END_STATE
STATE delay3s
        TIMEOUT T#3s THEN
                open0 := FALSE;
                open1 := FALSE;
                open2 := FALSE;
                SET NEXT;
        END_TIMEOUT
END_STATE
STATE check_closed
        IF      (door0closed AND
                 door1closed AND
                 door2closed) THEN
                STOP;
        END_IF
END_STATE
END_PROCESS
END_PROGRAM
(* Plant *)
PROGRAM Plant
VAR_INPUT
(* floor sensors *)
onfloor0 : BOOL;
onfloor1 : BOOL;
onfloor2 : BOOL;
(* floor call buttons *)
call0 : BOOL;
call1 : BOOL;
call2 : BOOL;
(* inside car call buttons *)
button0 : BOOL;
button1 : BOOL;
button2 : BOOL;
(* doors sensors *)
door0closed : BOOL;
door1closed : BOOL;
door2closed : BOOL;
END_VAR
VAR_OUTPUT
(* elevator up/down *)
up : BOOL;
down : BOOL;
(* doors open *)
open0 : BOOL;
```

```
open1 : BOOL;
open2 : BOOL;
(* LEDs for inside/outside call buttons *)
call0_LED : BOOL;
call1_LED : BOOL;
call2_LED : BOOL;
button0_LED : BOOL;
button1_LED : BOOL;
button2_LED : BOOL;
(* cur floor LEDs *)
floor0_LED : BOOL;
floor1_LED : BOOL;
floor2_LED : BOOL;
(* cur floor num *)
cur : INT;
END_VAR
PROCESS Init
STATE begin
        (* inputs: *)
        onfloor0 := FALSE;
        onfloor1 := FALSE;
        onfloor2 := FALSE;
        call0 := FALSE;
        call1 := FALSE;
        call2 := FALSE;
        button0 := FALSE;
        button1 := FALSE;
        button2 := FALSE;
        door0closed := FALSE;
        door1closed := FALSE;
        door2closed := FALSE;
        (* outputs: *)
        up := FALSE;
        down := FALSE;
        open0 := FALSE;
        open1 := FALSE;
        open2 := FALSE;
        call0_LED := FALSE;
        call1_LED := FALSE;
        call2_LED := FALSE;
        button0_LED := FALSE;
        button1_LED := FALSE;
        button2_LED := FALSE;
        floor0_LED := FALSE;
        floor1_LED := FALSE;
        floor2_LED := FALSE;

        START PROCESS Door0Sim;
        START PROCESS Door1Sim;
        START PROCESS Door2Sim;
        START PROCESS ElevatorSim;
        STOP;
END_STATE
END_PROCESS
PROCESS Door0Sim
VAR CONSTANT
        DOOR_SPEED : REAL := 0.5;
        DOOR_OPEN_COORD : REAL := -50;
END_VAR
VAR
        coord : REAL := 0.0;
END_VAR
STATE check_open_close LOOPED
        IF open0 THEN
                coord := coord - DOOR_SPEED;
        ELSE
                coord := coord + DOOR_SPEED;
        END_IF
```

```
        IF coord >= 0.0 THEN
                coord := 0.0;
        END_IF
        IF coord <= DOOR_OPEN_COORD THEN
                coord := DOOR_OPEN_COORD;
        END_IF
        IF coord = 0.0 THEN
                door0closed := TRUE;
        ELSE
                door0closed := FALSE;
        END_IF
END_STATE
END_PROCESS
PROCESS Door1Sim
VAR CONSTANT
        DOOR_SPEED : REAL := 0.5;
        DOOR_OPEN_COORD : REAL := -50;
END_VAR
VAR
        coord : REAL := 0.0;
END_VAR
STATE check_open_close LOOPED
        IF open1 THEN
                coord := coord - DOOR_SPEED;
        ELSE
                coord := coord + DOOR_SPEED;
        END_IF
        IF coord >= 0.0 THEN
                coord := 0.0;
        END_IF
        IF coord <= DOOR_OPEN_COORD THEN
                coord := DOOR_OPEN_COORD;
        END_IF
        IF coord = 0.0 THEN
                door1closed := TRUE;
        ELSE
                door1closed := FALSE;
        END_IF
END_STATE
END_PROCESS
PROCESS Door2Sim
VAR CONSTANT
        DOOR_SPEED : REAL := 0.5;
        DOOR_OPEN_COORD : REAL := -50;
END_VAR
VAR
        coord : REAL := 0.0;
END_VAR
STATE check_open_close LOOPED
        IF open2 THEN
                coord := coord - DOOR_SPEED;
        ELSE
                coord := coord + DOOR_SPEED;
        END_IF
        IF coord >= 0.0 THEN
                coord := 0.0;
        END_IF
        IF coord <= DOOR_OPEN_COORD THEN
                coord := DOOR_OPEN_COORD;
        END_IF
        IF coord = 0.0 THEN
                door2closed := TRUE;
        ELSE
                door2closed := FALSE;
        END_IF
END_STATE
END_PROCESS
PROCESS ElevatorSim
```

```
VAR CONSTANT
        ELEV_ACCEL : REAL := 0.25;
        ELEV_MAX_SPEED : REAL := 0.5;
        ELEV_DOWN_COORD : REAL := 440.0;
END_VAR
VAR
        v : REAL := 0.0;
        coord : REAL := 0.0;
END_VAR
STATE up_down LOOPED
        IF up THEN (* velocity *)
                v := v - ELEV_ACCEL;
        ELSIF down THEN
                v := v + ELEV_ACCEL;
        ELSE
                v := 0.0;
        END_IF
        IF v > ELEV_MAX_SPEED THEN
                v := ELEV_MAX_SPEED;
        ELSIF v < 0 - ELEV_MAX_SPEED THEN
                v := 0 - ELEV_MAX_SPEED;
        END_IF
        coord := coord + v; (* coordinate *)
        IF coord < 0.0 THEN
                coord := 0.0;
        ELSIF coord > ELEV_DOWN_COORD THEN
                coord := ELEV_DOWN_COORD;
        END_IF

        onfloor0 := FALSE; (* sensors *)
        onfloor1 := FALSE;
        onfloor2 := FALSE;
        IF coord < 1.5 THEN
                onfloor2 := TRUE;
        ELSIF (coord > 224.5) AND (coord < 225.5) THEN
                onfloor1 := TRUE;
        ELSIF (coord > ELEV_DOWN_COORD - 20.0) THEN
                onfloor0 := TRUE;
        END_IF
END_STATE
END_PROCESS
END_PROGRAM
```

Программа HandDryer:

```
CONFIGURATION Conf
RESOURCE Res1 ON TestCPU
        TASK T1 (INTERVAL := T#100ms, PRIORITY := 1);
        PROGRAM dryer WITH T1 : HandDryer;
END_RESOURCE
END_CONFIGURATION
PROGRAM HandDryer
   VAR_INPUT
       hands : BOOL;
   END_VAR
   VAR_OUTPUT
       control : BOOL;
   END_VAR
   PROCESS Dry
       STATE Wait
           IF hands THEN
               control := TRUE;
               SET NEXT;
           END_IF
       END_STATE
```

```
        STATE Work
            IF hands THEN
                RESET TIMER;
            END_IF
            TIMEOUT T#2s THEN
                control := FALSE;
                SET STATE Wait;
            END_TIMEOUT
        END_STATE
    END_PROCESS
END_PROGRAM
```

Приложение С.

Результаты работу модуля для тестовых программ.

Граф DemoProject:

```
digraph CFG {
0 [label="id 0 Process MainProcess"];
1 [label="id 1 State Init"];
2 [label="id 2 IF  input1"];
3 [label="id 3 SET NEXT"];
4 [label="id 4 State Execute"];
5 [label="id 5 IF  input2 > 10"];
6 [label="id 6 FOR output2:= 1 TO 3"];
7 [label="id 7 WHILE  input2 < 10"];
8 [label="id 8 REPEAT  output2 = 0"];
9 [label="id 9 IF  output1"];
10 [label="id 10 RETURN"];
11 [label="id 11 EXIT"];
12 [label="id 12 RESET TIMER"];
13 [label="id 13 SET_STATE"];
14 [label="id 14 State Idle"];
15 [label="id 15 TimeoutStatement"];
16 [label="id 16 SET STATE"];
17 [label="id 17 Process SecondaryProcess"];
18 [label="id 18 State Start"];
19 [label="id 19 IF  input2 > 5"];
20 [label="id 20 START PROCESS MainProcess"];
21 [label="id 21 SET_NEXT"];
22 [label="id 22 State Stop"];
23 [label="id 23 IF  input2 < 5"];
24 [label="id 24 ERROR PROCESS MainProcess"];
25 [label="id 25 STOP PROCESS MainProcess"];
0 -> 1 [label=""];
1 -> 2 [label=""];
2 -> 3 [label=""];
3 -> 4 [label=""];
0 -> 4 [label=""];
4 -> 5 [label=""];
5 -> 6 [label=""];
6 -> 7 [label=""];
7 -> 8 [label=""];
8 -> 9 [label=""];
9 -> 10 [label=""];
9 -> 11 [label="FALSE"];
10 -> 11 [label=""];
11 -> 12 [label=""];
12 -> 13 [label=""];
13 -> 14 [label=""];
0 -> 14 [label=""];
14 -> 15 [label=""];
15 -> 16 [label=""];
16 -> 1 [label=""];
17 -> 18 [label=""];
18 -> 19 [label=""];
19 -> 20 [label=""];
20 -> 0 [label=""];
19 -> 21 [label="FALSE"];
20 -> 21 [label=""];
21 -> 22 [label=""];
17 -> 22 [label=""];
22 -> 23 [label=""];
23 -> 25 [label=""];
25 -> 0 [label=""];
23 -> 24 [label="ELSE"];
24 -> 0 [label=""];
}
```

Граф Elevator:

```
digraph CFG {
0 [label="id 0 Process Init"];
1 [label="id 1 State begin"];
2 [label="id 2 START PROCESS Call0Latch"];
3 [label="id 3 START PROCESS Call1Latch"];
4 [label="id 4 START PROCESS Call2Latch"];
5 [label="id 5 START PROCESS Button0Latch"];
6 [label="id 6 START PROCESS Button1Latch"];
7 [label="id 7 START PROCESS Button2Latch"];
8 [label="id 8 START PROCESS CheckCurFloor"];
9 [label="id 9 START PROCESS UpControl"];
10 [label="id 10 STOP PROCESS "];
11 [label="id 11 Process Call0Latch"];
12 [label="id 12 State init"];
13 [label="id 13 SET_NEXT"];
14 [label="id 14 State check_ON_OFF"];
15 [label="id 15 IF  call0 &&  NOT prev_in"];
16 [label="id 16 IF  open0 &&  NOT prev_out"];
17 [label="id 17 Process Call1Latch"];
18 [label="id 18 State init"];
19 [label="id 19 SET_NEXT"];
20 [label="id 20 State check_ON_OFF"];
21 [label="id 21 IF  call1 &&  NOT prev_in"];
22 [label="id 22 IF  open1 &&  NOT prev_out"];
23 [label="id 23 Process Call2Latch"];
24 [label="id 24 State init"];
25 [label="id 25 SET_NEXT"];
26 [label="id 26 State check_ON_OFF"];
27 [label="id 27 IF  call2 &&  NOT prev_in"];
28 [label="id 28 IF  open2 &&  NOT prev_out"];
29 [label="id 29 Process Button0Latch"];
30 [label="id 30 State init"];
31 [label="id 31 SET_NEXT"];
32 [label="id 32 State check_ON_OFF"];
33 [label="id 33 IF  button0 &&  NOT prev_in"];
34 [label="id 34 IF  open0 &&  NOT prev_out"];
35 [label="id 35 Process Button1Latch"];
36 [label="id 36 State init"];
37 [label="id 37 SET_NEXT"];
38 [label="id 38 State check_ON_OFF"];
39 [label="id 39 IF  button1 &&  NOT prev_in"];
40 [label="id 40 IF  open1 &&  NOT prev_out"];
41 [label="id 41 Process Button2Latch"];
42 [label="id 42 State init"];
43 [label="id 43 SET_NEXT"];
44 [label="id 44 State check_ON_OFF"];
45 [label="id 45 IF  button2 &&  NOT prev_in"];
46 [label="id 46 IF  open2 &&  NOT prev_out"];
47 [label="id 47 Process CheckCurFloor"];
48 [label="id 48 State check_floor"];
49 [label="id 49 IF  onfloor0"];
50 [label="id 50 Process UpControl"];
51 [label="id 51 State check_calls"];
52 [label="id 52 IF  (cur = 0 && (call0_LED || button0_LED)) || (cur = 1 && (call1_LED ||
button1_LED)) || (cur = 2 && (call2_LED || button2_LED))"];
53 [label="id 53 CaseStatement"];
54 [label="id 54 START PROCESS DoorCycle"];
55 [label="id 55 SET_STATE"];
56 [label="id 56 State check_stop"];
57 [label="id 57 IF  ( )"];
58 [label="id 58 START PROCESS DoorCycle"];
59 [label="id 59 SET_NEXT"];
60 [label="id 60 State door_cycle"];
61 [label="id 61 IF  ( )"];
62 [label="id 62 START PROCESS "];
63 [label="id 63 Process UpMotion"];
```

```
64 [label="id 64 State start"];
65 [label="id 65 CaseStatement"];
66 [label="id 66 State check_target"];
67 [label="id 67 IF  (cur = target)"];
68 [label="id 68 STOP PROCESS "];
69 [label="id 69 Process DownControl"];
70 [label="id 70 State check_calls"];
71 [label="id 71 IF  (cur = 0 && (call0 || button0)) || (cur = 1 && (call1 || button1)) ||
(cur = 2 && (call2 || button2))"];
72 [label="id 72 CaseStatement"];
73 [label="id 73 START PROCESS DoorCycle"];
74 [label="id 74 SET_STATE"];
75 [label="id 75 State check_stop"];
76 [label="id 76 IF  ( )"];
77 [label="id 77 START PROCESS DoorCycle"];
78 [label="id 78 SET_NEXT"];
79 [label="id 79 State door_cycle"];
80 [label="id 80 IF  ( )"];
81 [label="id 81 START PROCESS "];
82 [label="id 82 Process DownMotion"];
83 [label="id 83 State start"];
84 [label="id 84 CaseStatement"];
85 [label="id 85 State chech_next"];
86 [label="id 86 IF  (cur = target)"];
87 [label="id 87 STOP PROCESS "];
88 [label="id 88 Process DoorCycle"];
89 [label="id 89 State choose_door_to_open"];
90 [label="id 90 CaseStatement"];
91 [label="id 91 SET_NEXT"];
92 [label="id 92 State delay3s"];
93 [label="id 93 TimeoutStatement"];
94 [label="id 94 SET_NEXT"];
95 [label="id 95 State check_closed"];
96 [label="id 96 IF  (door0closed && door1closed && door2closed)"];
97 [label="id 97 STOP PROCESS "];
0 -> 1 [label=""];
1 -> 2 [label=""];
2 -> 11 [label=""];
2 -> 3 [label=""];
3 -> 17 [label=""];
3 -> 4 [label=""];
4 -> 23 [label=""];
4 -> 5 [label=""];
5 -> 29 [label=""];
5 -> 6 [label=""];
6 -> 35 [label=""];
6 -> 7 [label=""];
7 -> 41 [label=""];
7 -> 8 [label=""];
8 -> 47 [label=""];
8 -> 9 [label=""];
9 -> 50 [label=""];
9 -> 10 [label=""];
11 -> 12 [label=""];
12 -> 13 [label=""];
13 -> 14 [label=""];
11 -> 14 [label=""];
14 -> 15 [label=""];
15 -> 16 [label=""];
17 -> 18 [label=""];
18 -> 19 [label=""];
19 -> 20 [label=""];
17 -> 20 [label=""];
20 -> 21 [label=""];
21 -> 22 [label=""];
23 -> 24 [label=""];
24 -> 25 [label=""];
25 -> 26 [label=""];
```

```
23 -> 26 [label=""];
26 -> 27 [label=""];
27 -> 28 [label=""];
29 -> 30 [label=""];
30 -> 31 [label=""];
31 -> 32 [label=""];
29 -> 32 [label=""];
32 -> 33 [label=""];
33 -> 34 [label=""];
35 -> 36 [label=""];
36 -> 37 [label=""];
37 -> 38 [label=""];
35 -> 38 [label=""];
38 -> 39 [label=""];
39 -> 40 [label=""];
41 -> 42 [label=""];
42 -> 43 [label=""];
43 -> 44 [label=""];
41 -> 44 [label=""];
44 -> 45 [label=""];
45 -> 46 [label=""];
47 -> 48 [label=""];
48 -> 49 [label=""];
50 -> 51 [label=""];
51 -> 52 [label=""];
52 -> 54 [label=""];
54 -> 88 [label=""];
54 -> 55 [label=""];
55 -> 60 [label=""];
50 -> 56 [label=""];
56 -> 57 [label=""];
57 -> 58 [label=""];
58 -> 88 [label=""];
58 -> 59 [label=""];
59 -> 60 [label=""];
50 -> 60 [label=""];
60 -> 61 [label=""];
61 -> 62 [label=""];
63 -> 64 [label=""];
63 -> 66 [label=""];
66 -> 67 [label=""];
67 -> 68 [label=""];
69 -> 70 [label=""];
70 -> 71 [label=""];
71 -> 73 [label=""];
73 -> 88 [label=""];
73 -> 74 [label=""];
74 -> 79 [label=""];
69 -> 75 [label=""];
75 -> 76 [label=""];
76 -> 77 [label=""];
77 -> 88 [label=""];
77 -> 78 [label=""];
78 -> 79 [label=""];
69 -> 79 [label=""];
79 -> 80 [label=""];
80 -> 81 [label=""];
82 -> 83 [label=""];
82 -> 85 [label=""];
85 -> 86 [label=""];
86 -> 87 [label=""];
88 -> 89 [label=""];
90 -> 91 [label=""];
91 -> 92 [label=""];
88 -> 92 [label=""];
92 -> 93 [label=""];
93 -> 94 [label=""];
94 -> 95 [label=""];
```

```
88 -> 95 [label=""];
95 -> 96 [label=""];
96 -> 97 [label=""];
}
```

Граф HandDryer:

```
digraph CFG {
0 [label="id 0 Process Dry"];
1 [label="id 1 State Wait"];
2 [label="id 2 IF  hands"];
3 [label="id 3 SET_NEXT"];
4 [label="id 4 State Work"];
5 [label="id 5 IF  hands"];
6 [label="id 6 RESET TIMER"];
7 [label="id 7 TimeoutStatement"];
8 [label="id 8 SET_STATE"];
0 -> 1 [label=""];
1 -> 2 [label=""];
2 -> 3 [label=""];
3 -> 4 [label=""];
0 -> 4 [label=""];
4 -> 5 [label=""];
5 -> 6 [label=""];
4 -> 7 [label=""];
7 -> 8 [label=""];
8 -> 1 [label=""];
}
```

Таблица 4 – значения метрик тестовых программ.

Название программы	Полученные значения метрик в формате Json

DemoProject	"LinesOfCodeMetric": {
	"linesCount": 95
	},
	"CyclomaticMetric": {
	"cyclomaticComplexity": 9
	},
	"PostSpecificMetric": {
	"processesCount": 2,
	"statesCount": 5,
	"statementsCount": 25,
	"averageStatesCount": 2.5,
	"averageStatementsCount": 5,
	"processTransitionsCount": 3
	}

Elevator	"LinesOfCodeMetric": {
	"linesCount": 589
	},
	"CyclomaticMetric": {
	"cyclomaticComplexity": 8
	},
	"PostSpecificMetric": {

	"processesCount": 13, "statesCount": 27, "statementsCount": 132, "averageStatesCount": 2.076, "averageStatementsCount": 4.88, "processTransitionsCount": 30 }

HandDryer	"LinesOfCodeMetric": {
	"linesCount": 44
	},
	"CyclomaticMetric": {
	"cyclomaticComplexity": 3
	},
	"PostSpecificMetric": {
	"processesCount": 1,
	"statesCount": 2,
	"statementsCount": 7,
	"averageStatesCount": 2,
	"averageStatementsCount": 3.5,
	"processTransitionsCount": 0
	}

Приложение Д.

Модуль статического анализа для программ на языке poST.

Руководство оператора

Листов 7

Новосибирск, 2024.

Содержание

Аннотация

В данном программном документе приведено руководство оператора для модуля статического анализа программ на языке poST. Исходными языками программы являются Java и Xtend. Средство разработки – IDE Eclipse, вспомогательное средство – среда разработки DSL Xtext.

Основной функцией программы является подсчет программных метрик кода и их сохранение в отдельные файлы.

Оформление программного документа «Руководство оператора» выполнено в соответствии с ГОСТ 19.505-79 «ЕСПД. Руководство оператора» и ГОСТ 19.105-78 «ЕСПД. Общие требования к программным документам».

1 Назначение программы

1.1 Основные сведения

Язык poST ориентирован на создание управляющих алгоритмов в промышленной автоматизации. Этот язык появился недавно и поэтому имеется острая потребность в анализе написанного кода. Так как не всегда существует уже изготовленный реальный объект управления, на котором можно провести тестирования, то предлагается применять статический анализ без запуска написанной программы.

1.2 Функции, выполняемые программой

Программа позволяет редактировать текст программ на poST. При сохранении файла с исходным кодом программа считает определенный набор значений метрик, а также строит граф потока управления для исходной программы на языке poST и генерирует файлы, в которых лежат соответствующие значения и граф в формате DOT.

2 Условия выполнения программы

2.1 Установка необходимых средств разработки

Перед запуском программы оператор должен выполнить следующие шаги:

1. Установить IDE Eclipse на компьютер.
2. Установить расширение Xtext для Eclipse IDE.
3. Установить расширение Xtend для Eclipse IDE.
4. Провести импорт ядра языка poST для Eclipse IDE.

3 Выполнение программы

3.1 Предварительные действия

Перед тем, как приступить к работе с программой на языке poST, оператор должен выполнить следующее:

1. Щелкнуть правой кнопкой мыши по папке su.nsk.iae.post.StatAnalyz в разделе Package Explorer;
2. Выбрать Run As – Eclipse Application;
3. В появившемся окне нажать кнопку Continue;
4. После выполнения третьего шага должно открыть новое окно runtime-Eclipse, в нем оператор должен создать новый проект: File – New – Project – PoST Project – нажать Next – ввести имя проекта и нажать Finish.

3.2 Запуск редактора кода

Далее оператор должен создать новый файл: щелкнуть правой кнопкой мыши по папке src в разделе Package Explorer – выбрать New – PoST file – ввести имя файла – нажать Finish.

3.3 Запуск модуля

Для запуска модуля оператор должен сохранить редактируемый файл с расширением post сочетанием клавиш ctrl+s, либо через меню File – Save. В случае, если файл не содержит ошибок, модуль создаст новые файлы в папке src-gen/ проекта, созданного ранее.

4 Сообщения оператору

Если файл с кодом на языке poST содержит ошибки, редактор кода указывает на проблемные места красным подчеркиванием. При наведении курсора на эти участки отображается сообщение с информацией о синтаксической ошибке.