МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

Образовательная программа: 09.03.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

РЕФАКТОРИНГ ТРАНСЛЯТОРА ЯЗЫКА REFLEX НА ОСНОВЕ АВТОМАТИЧЕСКОЙ ПАРСЕР-ГЕНЕРАЦИИ

утверждена распоряжением проректора по учебной работе №___ от «__» _____ 20__г.

скорректирована распоряжением проректора по учебной работе №___ от «__» _____ 20__г.

Бастрыкина Алена Алексеевна, группа 16203 _____

(Фамилия, Имя, Отчество студента, группа) (подпись студента)

«К защите допущена»

Заведующий кафедрой КТ,
д.т.н., доцент
Зюбин В.Е. / ……..

(ФИО) / (подпись)

«……»……2020г.

Руководитель ВКР

д.т.н., доцент, в.н.с.
зав. каф. КТ ФИТ НГУ
Зюбин В.Е. / ……..

(ФИО) / (подпись)

«……»……2020г.

Соруководитель ВКР

м.н.с., ст. препод. каф. КТ ФИТ НГУ
Розов А.С. / ……..

(ФИО) / (подпись)

«……»……2020г.

Дата защиты: «……»……2020г.

Новосибирск, 2020г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

УТВЕРЖДАЮ
Зав. кафедрой Зюбин В.Е.

…….....
(подпись)

«15» сентября 2019г.

Студенту Бастрыкиной Алене Алексеевне, группы 16203

Тема «Рефакторинг транслятора языка Reflex на основе автоматической парсер-генерации»

утверждена распоряжением проректора по учебной работе № 0290 от «30» сентября 2019г.

Срок сдачи студентом готовой работы «__» _____ 2020 г.

Исходные данные (или цель работы): модификация транслятора языка Reflex с использованием средств автоматической парсер-генерации

Структурные части работы: анализ предметной области, выбор средств парсер-генерации, программная реализация, апробация на тестовой задаче

Консультанты по разделам ВКР отсутствуют

Руководитель ВКР

доцент, д.т.н.

Зюбин В. Е. / ……..

«15» сентября 2019г.

Задание принял к исполнению

Бастрыкина А. А. / ……..

«15» сентября 2019г.

СОДЕРЖАНИЕ

СОКРАЩЕНИЯ, ОБОЗНАЧЕНИЯ..... 3

ВВЕДЕНИЕ..... 4

1 Анализ предметной области и формулировка требований к разрабатываемым средствам..... 6

1.1 Анализ специфики языка Reflex..... 6
1.2 Инструментальные средства для языка Reflex..... 7
1.3 Недостатки существующих средств..... 7
1.4 Требования к разрабатываемым инструментальным средствам..... 8

2 Синтаксис и семантика языка Reflex. Выбор средств парсер-генерации и проектирование архитектуры системы..... 8

2.1 Синтаксис языка Reflex..... 8
2.2 Семантика трансформаций из языка Reflex в Си..... 12
2.2.1 Структура программы (гиперпроцесс, процессы)..... 12
2.2.2 Управляющие конструкции..... 15
2.2.3 Типы..... 16
2.2.4 Порты и переменные..... 16
2.2.5 Временные интервалы..... 18
2.2.6 Аннотации..... 18
2.3 Подходы к созданию предметно-ориентированных языков..... 18
2.4 Обзор и сравнительный анализ средств парсер-генерации и создания предметно-ориентированных языков..... 19
2.5 Архитектура разрабатываемой программной системы..... 22
2.5.1 Компоненты разрабатываемой программной системы и их взаимодействие..... 22
2.5.2 Структура модулей системы..... 23

3 Реализация инструментальных средств для языка Reflex и их апробация на тестовой задаче..... 24

3.1 Внутреннее представление программы..... 24
3.2 Описание грамматики языка Reflex в нотации Xtext..... 25
3.3 Реализация модуля кодогенерации..... 27
3.3.1 Структура генерируемых файлов..... 27
3.3.2 Структура модуля кодогенерации и алгоритм кодогенерации..... 29
3.3.3 Генерация кода под платформу Arduino..... 32
3.4 Семантический анализ кода на языке Reflex средствами Xtext..... 34
3.5 Редактор кода..... 37
3.6 Апробация разработанных средств..... 38

ЗАКЛЮЧЕНИЕ..... 39

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ..... 40

ПРИЛОЖЕНИЕ А. Грамматика Reflex в нотации Xtext..... 43

ПРИЛОЖЕНИЕ В. Диаграмма классов модуля кодогенерации..... 50

ПРИЛОЖЕНИЕ С. Набор семантических проверок для языка Reflex..... 51

ПРИЛОЖЕНИЕ D. Листинг программы на языке Reflex для задачи «Сушилка для рук»..... 53

СОКРАЩЕНИЯ, ОБОЗНАЧЕНИЯ

IDE – Integrated Development Environment

AST – Abstract Syntax Tree

DSL – domain-specific language

РБНФ – расширенная форма Бэкуса-Наура

LSP (Language Server Protocol) – протокол, разработанный компанией Microsoft с целью сокращения затрат на разработку поддержки языков под разные IDE. LSP определяет контракт между Language Server (его реализацией для конкретного языка программирования) и Language Client (редактором кода или IDE)

RIDE 2.0 (Reflex IDE 2.0) – Интегрированная среда разработки для языка Reflex, разрабатываемая студентами и сотрудниками лаборатории 19 Института автоматики и электрометрии СО РАН

ВВЕДЕНИЕ

Использование языков общего назначения для программирования цифровых систем управления ведет к росту сложности программной архитектуры и алгоритмов, увеличению стоимости разработки и затрудняет поддержку таких систем.

Для решения этой проблемы в Институте автоматики и электрометрии СО РАН был разработан процесс-ориентированный язык Reflex [1]. В настоящее время планируется создание RIDE 2.0 – IDE для языка Reflex с возможностью интеграции дополнительных компонентов для анализа и преобразования программ на Reflex, например: интерактивный редактор кода, модули кодогенерации, динамической и статической верификации.

Для языка Reflex был разработан транслятор в процедурный язык C. Парсер этого транслятора был написан на языке C по грамматике языка Reflex. Такая реализация затрудняет его сопровождение и препятствует модификации устаревшего синтаксиса языка. Помимо этого, транслятор обладает рядом архитектурных недостатков, которые делают его непригодным для интеграции в RIDE 2.0.

Цель работы – разработка транслятора языка Reflex с использованием методов автоматической парсер-генерации.

Для достижения этой цели были поставлены следующие задачи:

1. Анализ специфики языка Reflex, области его применения, существующих инструментальных средств.
2. Формулировка требований к разрабатываемым языковым и инструментальным средствам.
3. Коррекция синтаксиса и семантических правил языка Reflex.
4. Определение трансформационной семантики для языка Си с учетом особенностей целевой платформы.
5. Выбор средств парсер-генерации.
6. Проектирование архитектуры транслятора.
7. Реализация парсера и семантического анализатора.
8. Реализация кодогенератора для тестовой целевой платформы.
9. Практическая апробация созданных инструментальных средств на тестовой задаче.

Использование инструментов парсер-генерации при разработке транслятора языка Reflex сократит затраты на его сопровождение, а также облегчит процесс модификации и расширения синтаксиса языка. Рефакторинг архитектуры транслятора и использование современных средств разработки позволит интегрировать его в RIDE 2.0, а также обеспечить

расширяемость – возможность разработки множества кодогенераторов на основе созданного парсера.

Работа изложена в трех главах. В первой главе анализируется специфика языка Reflex, существующих инструментальных средств и формулируются требования к разрабатываемым языковым и инструментальным средствам. Во второй главе описываются откорректированные синтаксис и семантика языка Reflex, в том числе трансформационная семантика для языка Си, выбранные средства парсер-генерации и архитектура транслятора. Третья глава посвящена вопросам реализации спроектированных инструментальных средств и их практической апробации на тестовой задаче.

1 Анализ предметной области и формулировка требований к разрабатываемым средствам

1.1 Анализ специфики языка Reflex

Процесс-ориентированный язык Reflex, созданный как диалект языка Си, предназначен для программирования систем управления в области промышленной автоматизации, основанных на программируемых логических микроконтроллерах (ПЛК). Отличительными особенностями таких систем являются: наличие объекта управления (*открытость*), непрерывное воздействие на объект управления со стороны управляющего алгоритма, определяемое происходящими на объекте управления событиями (*цикличность*), синхронизация исполнения управляющего алгоритма с физическими процессами на объекте управления (*синхронизм*), а также *логический параллелизм* – протекание на объекте управления множества независимых процессов. [2]

Язык Reflex обеспечивает набор средств, необходимый для описания алгоритмов, обладающих такими особенностями. В основе концепции языка лежит модель гиперпроцесса, описанная в статье [3]. Так, программа на языке Reflex – это упорядоченное множество процессов, которые представляют собой конечные автоматы с некоторым набором состояний и множеством правил перехода между этими состояниями. Модель предполагает циклический запуск множества процессов с заданным периодом активизации. Во время работы процессы меняют своё текущее состояние в соответствии со внешними событиями на объекте управления, либо в соответствии со временными событиями (тайм-аутами). Язык предоставляет средства для организации взаимодействия процессов: операторы запуска и остановки процессов, переменные, разделяемые между процессами. В языке Reflex существует спецификация для удобного описания входных и управляющих сигналов, которые привязаны к физическим портам управляющего устройства (ПЛК), а также предоставляются средства для чтения и изменения значений управляющих сигналов (переменные, отображаемые на входные и выходные порты).

Было определено алгоритмически эквивалентное преобразование языка Reflex в компилируемый язык Си. Язык был отработан на реальных задачах в рамках масштабных проектов и доказал свою применимость и эффективность. [4]

1.2 Инструментальные средства для языка Reflex

Помимо самой концепции языка, в Институте автоматики и электрометрии СО РАН был разработан ряд инструментальных средств для обработки, анализа и трансляции программ на Reflex.

Для языка Reflex на данный момент реализованы трансляторы, преобразующие код на языке Reflex в Си (R2C), в Python (R2Py) и в язык Formula Node пакета LabVIEW. [2] Парсеры трансляторов базируются на LL(1)-алгоритме синтаксического разбора. Трансляторы осуществляют семантический анализ кода и предоставляют пользователю локализованные сообщения об ошибках.

Было создано расширение для Notepad++, предназначенное для работы с исходными кодами на языке Reflex, реализующее базовую функциональность редактирования файлов и запуска транслятора. [5]

В настоящее время ведется работа над созданием IDE для языка Reflex (RIDE 2.0). К IDE было выдвинуто основное требование: расширяемость – возможность интеграции отладчика, множества модулей верификации программ и кодогенерации.

1.3 Недостатки существующих средств

В результате исследования существующих инструментальных средств было обнаружено, что транслятор R2C устарел и является непригодным для интеграции в IDE, так как он обладает рядом недостатков:

- Парсер языка реализован «вручную», без использования средств автоматической парсер-генерации, несмотря на то, что грамматика языка подходит для описания в нотациях, которую используют средства парсер-генерации (БНФ, РБНФ), поскольку существует описание грамматики языка в РБНФ [6]. Такой подход затрудняет сопровождение транслятора, особенно в случае, когда появляется необходимость изменения синтаксиса языка;

  Решение: ипользование средств автоматической парсер-генерации

- Парсер, семантический анализатор и кодогенератор составляют один программный модуль. Это противоречит требованиям к архитектуре IDE, а также влечет за собой существенный недостаток: реализация нового кодогенератора возможна лишь посредством переиспользования и модификации существующего кода.

Решение: разбиение транслятора на два модуля: модуль, включающий парсер и семантический анализатор и модуль кодогенерации.

- Решение написано на языке Си и требует компиляции под разные платформы.

Решение: выбор кроссплатформенных средств для разработки

- Работа с платформозависимыми кодировками (windows-1251), что в некоторых случаях является преградой для использования транслятора.

Решение: использование общепринятных кодировок (ASCII, UTF-8)

1.4 Требования к разрабатываемым инструментальным средствам

В результате анализа предметной области к разрабатываемым средствам были выдвинуты следующие требования:

- парсер должен работать с текстовыми файлами в кодировках ASCII;
- парсер должен распознавать англоязычный синтаксис языка со словами в нижнем регистре;
- модуль кодогенерации должен быть независим от модуля парсера;
- взаимодействие парсера и кодогенератора для целевой платформы должно осуществляться посредством передачи AST;
- пользователю должна предоставляться информация о семантических и синтаксических ошибках в коде, должна осуществляться привязка диагностических сообщений к номеру строки исходного кода;
- все реализованные средства должен быть кроссплатформенными.

2 Синтаксис и семантика языка Reflex. Выбор средств парсер-генерации и проектирование архитектуры системы

2.1 Синтаксис языка Reflex

Для описания был выбран англоязычный вариант языка с ключевыми словами в нижнем регистре. Синтаксис, определенный в [6] был переработан: был добавлен ряд новых синтаксических конструкций (временные интервалы, аннотации), часть старых конструкций были модифицированы.

Описание программы начинается с ключевого слова **program**. Тело программы заключается в фигурные скобки.

В теле программы сначала определяется период активизации процессов. Это делается помощью ключевого слова **clock** <*интервал*>**.** Для описания временных интервалов в язык были добавлены временные литералы – литералы вида **0t**<*число_дней*>**d**<*число_часов*>**h**<*число_минут*>**m**<*число_секунд*>**s**<*число_миллисекунд*>**ms**. Для указания периода активизации процессов могут быть использованы как временные интервалы, так и численные литералы (в этом случае число будет означать количество миллисекунд).

Далее определяется список констант, перечислений, глобальных переменных, портов ввода/вывода, список функций, доступных для вызова внутри тел процессов (Листинг 1).

Для описания входных и выходных портов используется последовательность <*тип_порта*> <*имя_порта*> <*адрес1*> <*адрес2*> <*разрядность*>, при этом для указания типа порта используются ключевые слова **input** и **output,** *адрес1* и *адрес2* – это два целочисленных значения, определяющих системный адрес порта, разрядность порта – это целое число, равное 8 или 16.

```
program Example {
    /* Период активизации процессов */
    clock 0t12ms;

    /* Описания констант */
    const bool ON = true;
    const bool OFF = false;
    const float PI = 3.14;
    const uint8 COUNT = 100;
    const time SECOND = 0t1s;

    /* Описания входных и выходных портов */
    input  in_port 0x00 0x01 8;
    output out_port 0x00 0x02 16;
    /* Сигнатуры доступных Си-функций */
    int8 SendMsgFloatParamGUI(int32, float);
    int8 SendMsgIntParamGUI(int32, int32);
    /* Описания глобальных переменных */
    bool in_signal = in_port[0];
    int16 out_signal = out_port[];
    int8 current_level;
    /* Пример перечисления */
    enum colors { red=0, yellow, green }
    <Описания процессов>
}
```

Листинг 1 – Описание атрибутов программы, общих для всех процессов

Описание сигнатур Си-функций производится аналогично описанию сигнатур функций в Си.

Константы описываются с помощью конструкции **const** *<тип_константы>* *<имя_константы>* **=** *<значение_константы>*. Указывается один из типов, которые используются в языке Рефлекс (**bool** – логический, **int8, int16, int32, int64, uint8, uint16, uint32, uint64 –** целые типы, **time** – временной тип, **float, double** - вещественные), идентификатор константы и ее значение. Значением константы может быть константное выражение.

Альтернативным способом описания констант является задание перечисления (**enum**). Спецификация перечисления в Reflex полностью совпадает со спецификацией перечислений в Си.

Существует возможность описания глобальных переменных, которые доступны всем процессам программы.

Глобальные переменные описываются с помощью конструкции *<тип_переменной>* *<имя переменной>*. Переменные не инициализируются при описании, однако существует возможность отображения переменных на физические порты. В этом случае после описания переменной через знак «=» указывается имя порта, а затем, опционально, в квадратных скобках, номер необходимого бита порта для получения логических значений. Если номер бита не указан, это означает, что порт отображается на переменную целиком.

После описания атрибутов программы идет описание процессов. Синтаксически процесс представляет собой структуру, начинающуюся с ключевого слова **process,** затем указывается идентификатор процесса и в фигурных скобках описывается тело процесса.

Тело процесса состоит из описания локальных и разделяемых переменных, а также спецификаций состояний процесса (листинг 2).

```
process Dryer {
    bool hands_under_dryer = in_port[0];
    bool dryer_control = out_port[0];
    /* Описание состояния ожидания */
    state Wait {
          if (hands_under_dryer) {
                dryer_control = ON;
                set state Work;
          }
    }
    /* Описание состояния сушки */
    state Work {
          if (hands_under_dryer) reset timer;
          timeout (SECOND) {
                dryer_control = OFF;
                set state Wait;
          }
    }
}
```

Листинг 2 – Пример описания процесса с состояниями, реализующего алгоритм управления сушилкой.

Описание локальных переменных производится аналогично описанию глобальных, однако описание переменной процесса может быть дополнено ключевым словом **shared** (листинг 3). Это даст возможность использования ее в других процессах. Для использования разделяемой переменной, которая объявлена в другом процессе, используется конструкция **shared** <*имя 1*>, <*имя 2*>, ..., <*имя N*> **from process** <*имя процесса*>.

```
process p1 {
    int my_shared_var shared;
}
process p2 {
    shared my_shared_var from process p1;
}
```

Листинг 3 – Описание и использование переменных, разделяемых между процессами.

Для описания состояния используется конструкция **state** <*имя состояния*> { <*тело состояния*> }. После имя состояния может быть добавлен модификатор **looped**, указывающий на то, что отсутствие переходов в другие состояния в теле этого состояния является контролируемым. Отсутствие модификатора **looped** для таких состояний приводит к семантической ошибке.

Тело состояний представляет собой последовательность выражений. В теле состояния с помощью конструкции **timeout** <*интервал*> { <*последовательность выражений*> } может быть описана последовательность выражений, выполняющихся по достижению времени нахождения процесса в состоянии указанного интервала.

Основные управляющие конструкции, такие как оператор условного перехода (**if-else**), оператор выбора (**switch-case**), заимствованы из языка Си.

Конструкция цикла в языке Reflex отсутствует.

Грамматика языка Reflex поддерживает все выражения (expressions) языка Си, кроме:

- тернарного оператора, оператора sizeof и оператора ‘,’;
- выражений для работы с указателями;
- выражений для работы с элементами массива.

В языке поддерживаются численные литералы языка Си. Строковые литералы в языке отсутствуют.

В качестве расширения языка выражений в языке Reflex присутствует специальное условное выражение вида **process** <*имя_процесса*> **in state** <*active* | *inactive* | *error* | *stop*>, проверяющее состояние указанного процесса.

В Reflex присутствуют специфические утверждения (statements), обеспечивающие процесс-ориентированность:

1. **set state** <*имя_состояния*> или **set next state** – оператор перехода в состояние по имени, либо в следующее по порядку;
2. **start process** <*имя_процесса*>, **stop process** <*имя_процесса*>, **error process** <*имя_процесса*> для управления жизненным циклом процессов, а также редуцированные их формы **restart, stop, error,** подразумевающие использование относительно текущего процесса;
3. **reset timer** для сброса таймера процесса.

В язык были добавлены аннотации – конструкции вида [<*пространство имен*>.<*имя*> = "<*значение*>"] или сокращенная форма [<*имя*> = "<*значение*>"]. Аннотации могут быть объявлены перед описанием программы, процесса или состояния. При этом каждая из этих сущностей может быть помечена любым количеством аннотаций.

Язык поддерживает однострочные и многострочные комментарии. Формат однострочных комментариев – // <*комментарий*>, многострочных – /* <*комментарий*> */.

2.2 Семантика трансформаций из языка Reflex в Си

Семантику языка Reflex можно определить с помощью правил трансляции в язык Си. Основу правил трансформации для нового транслятора составляют правила трансформации и приемы алгоритмической реализации гиперпроцессов и процессов, которые были описаны в пособии [2], а также в статье [6]. Для новых и модифицированных синтаксических конструкций семантика была дополнена или изменена.

2.2.1 Структура программы (гиперпроцесс, процессы)

Реализация гиперпроцесса на языке Си представлена в листинге 4. Для программной реализации гиперпроцесса используется подход, при котором процессы представляют собой упорядоченный набор функций языка Си, вызывающихся в бесконечном цикле. Таким образом, процессы взаимодействуют друг с другом в рамках кооперативной многозадачности.

```
void main(void) {
    init_processes(); /* Инициализация массива процессов */
    init_time();      /* Инициализация счетчика времени */
    init_io();        /* Инициализация портов ввода-вывода */
    for (;;) {        /* Цикл гиперпроцесса */
          cur_time = get_time();
          if (cur_time >= next_act_time) { /* Проверка наступления
                                              момента активизации */
                /* Вычисление следующего момента активизации */
                next_act_time += INTERVAL
                if (next_act_time - cur_time > INTERVAL) {
                      /* Обработка ситуаций, когда ресурсоемкость
                         алгоритма      превышает INTERVAL */
                      next_act_time = cur_time + INTERVAL;
                }
                input();
                process1();
                process2();
                ...
                processN();
                output();
          }
    }
}
```

Листинг 4 – Реализация гиперпроцесса на языке Си

Для обеспечения синхронизованного запуска гиперпроцесса с заданной периодичностью и обеспечения работы оператора реакции на временные события (оператор **timeout)** используется набор следующих приемов:

- считывание и сохранение текущего значения таймера устройства в глобальную переменную на каждой итерации бесконечного цикла;
- вычисление времени следующей активации гиперпроцесса перед запуском гиперпроцесса и его сохранение в глобальную переменную. При вычислении предполагается, что время активизации возрастает на величину, равную периоду активизации. Это предположение может нарушаться в ситуациях, при которых время выполнения управляющего алгоритма превышает период активизации. Такие ситуации можно распознать, посчитав разность между возросшим значением времени активизации и текущим значением времени. В нормальных ситуациях эта разность не превышает период активизации. Иначе, время следующей активизации принимается равным значению текущего времени, увеличенному на период активизации;
- условием запуска гиперпроцесса при такой реализации является момент, когда текущее время превышает или равно вычисленному значению времени активизации.

Для обеспечения взаимодействия алгоритма с внешней средой перед вызовом функций-процессов в цикле программы вызывается функция `input()`, после – `output()`.

Эти функции выполняют работу, связанную с чтением и записью значений переменных, которые отображаются на входные и выходные порты.

Для хранения информации о процессах используется статический массив структур (листинг 5) языка Си со следующими полями:

- `state_start_time` – время последнего перехода процесса в какое-либо состояние, либо время последнего выполнения операции **reset timer**;
- `cur_state` – поле, которое хранит идентификатор текущего состояния процесса.

Идентификаторы состояний нормального останова и ошибки однозначно задаются в программе и равны для всех процессов.

```
struct Process {
    uint32_t state_start_time; /* Время последнего обновления
                                  состояния */
    uint8_t  cur_state;        /* Идентификатор текущего
                                     состояния */
};
Process processes[PROCESS_COUNT];
```

Листинг 5 – Структура, хранящая состояния процессов

Перед началом работы бесконечного цикла гиперпроцесса массив инициализируется так, что идентификаторы состояний всех процессов, кроме первого, становятся равными идентификатору состояния останова, а идентификатор состояния первого процесса равен нулю, то есть соответствует идентификатору его начального состояния. Таким образом, в начальный момент программы активен только один процесс и он находится в своем начальном состоянии.

Функции-процессы представляют собой программную реализацию конечного автомата через оператор **switch**. Пример такой функции представлен в листинге 6. Для каждого состояния процесса организуется блок **case**, опцией блока является идентификатор состояния. Тело блока **case** соответствует телу состояния – последовательности выражений и операторов.

```
void initial_process() { /* Процесс с идентификатором 0 */
    switch (processes[0].cur_state) {/* Проверка текущего
                                        состояния */
          case 0: { /* Состояние с идентификатором 0 –
                                        начальное */
                <Тело начального состояния>
                break;
          }
          case 1: { /* Состояние с идентификатором 1 */
                <Тело первого состояния>
                break;
          }
          ...
          case m: { /* Состояние с идентификатором m */
                <Тело m-того состояния>
                break;
          }
    }
```

Листинг 6 – Реализация процесса на языке Си

2.2.2 Управляющие конструкции

Конструкции языка Reflex, заимствованные из языка Си (оператор условного перехода **if-else**, оператор **switch-case,** арифметические и логические выражения), не претерпевают изменений при трансляции в язык Си. Интерес представляют конструкции, специфичные для языка Reflex. Правила трансформации для них представлены в таблице 1.

Конструкция языка Reflex	Правило трансляции
set state s;	Processes[i].cur_state = s_id; Processes[i].state_start_time = cur_time;
set next state;	Processes[i].cur_state += 1; Processes[i].state_start_time = cur_time;
start process p;	Processes[p_id].cur_state = 0; Processes[p_id].state_start_time = cur_time;
stop process p;	Processes[p_id].cur_state = STATE_OF_STOP; Processes[p_id].state_start_time = cur_time;
error process p;	Processes[p_id].cur_state = STATE_OF_ERROR; Processes[p_id].state_start_time = cur_time;
reset timer;	Processes[i].state_start_time = cur_time;
restart;	Processes[i].cur_state = 0; Processes[i].state_start_time = cur_time;
stop;	Processes[i].cur_state = STATE_OF_STOP; Processes[i].state_start_time = cur_time;
error;	Processes[i].cur_state = STATE_OF_ERROR; Processes[i].state_start_time = cur_time;

timeout <интервал> <последовательность выражений>	**if** (cur_time - Processes[i].state_start_time > <значение>) {
	<транслированная последовательность>
	}

process p **in state inactive**	Processes[i].cur_state == STATE_OF_STOP || Processes[i].cur_state == STATE_OF_ERROR
process p **in state**	Processes[i].cur_state != STATE_OF_STOP &&

active	`Processes[i].cur_state != STATE_OF_ERROR`
process p **in state error**	`Processes[i].cur_state == STATE_OF_ERROR`
process p **in state stop**	`Processes[i].cur_state == STATE_OF_STOP`

Таблица 1 – Правила трансляции выражений, специфичных для языка Reflex. *Пояснения к таблице: правила верны для для выражений, объявленных внутри процесса с идентификатором **i**. Идентификатор процесса **p** равен **p_id**, идентификатор состояния **s** равен **s_id**. **cur_time** – переменная, которая хранит время последней активизации гиперпроцесса, **STATE_OF_STOP**, **STATE_OF_ERROR** – целочисленные идентификаторы, соответствующие состоянию нормального останова и ошибки.*

2.2.3 Типы

Правила соответствия типов языка Reflex типам или определениям типов языка Си указаны в таблице 2.

Тип языка Reflex	Тип языка Си	Примечание
bool	char	Диапазон значений – 0 и 1
int8	int8_t	Определения типов взяты из стандартной библиотеки языка Си stdint.h
uint8	uint8_t	
int16	int16_t	
uint16	uint16_t	
int32	int32_t	
uint32	uint32_t	
int64	int64_t	
uint64	uint64_t	
time	uint32_t	
float	float	
double	double	

Таблица 2 – Правила трансформации для типов языка Reflex.

2.2.4 Порты и переменные

Объявления портов транслируются в объявления глобальных переменных языка Си. Переменные, соответствующие восьмиразрядным портам, имеют тип **int8_t**, шестнадцатиразрядным – **int16_t**.

Как было показано ранее, переменные в языке Reflex могут иметь привязку к портам. Будем называть такие переменные физическими, а переменные без привязки – программными. Переменная также имеет область видимости – глобальная, локальная и разделяемая.

При трансляции все переменные языка Reflex становятся глобальными переменными языка Си. Типы переменных транслируются согласно правилам соответствия типов.

Логика привязки значений физических переменных к значениям портов реализуется внутри описанных ранее функций `input()` и `output()`.

Для чтения и записи значений с физических портов ПЛК используются функции `read_byte`, `write_byte`, `read_word`, `write_word`, реализация которых зависит от аппаратной платформы. Для отображения значений битов портов на логические переменные программы используются битовые операции.

Примеры возможных определений портов и физических переменных, отображаемых на них, а также правила трансляции приведены в таблице 3.

Пример описания физической переменной	Правило трансляции

`input in_port 0x01 0x02 8;`
`int8 in_val = in_port[];`
	`int8_t in_port;`
	`int8_t in_val;`
	
	`void input() {`
	`  in_port = read_byte(0x01, 0x02);`
	`  in_val = in_port;`
	`}`

`input in_port 0x01 0x02 8;`
`bool in_signal = in_port[1];`
	`int8_t in_port;`
	`char in_signal;`
	
	`void input() {`
	`  in_port = read_byte(0x01, 0x02);`
	`  if (in_port & MASK1) {`
	`  in_signal = 1;`
	`  } else {`
	`  in_signal = 0;`
	`  }`
	`}`

`output out_port 0x01 0x02 8;`
`int8 out_val = out_port[];`
	`int8_t out_port;`
	`int8_t out_val;`
	
	`void output() {`
	`  out_port = out_val;`
	`  write_byte(0x01, 0x02, out_port);`
	`}`

`output out_port 0x01 0x02 8;`
`bool out_signal = out_port[1];`
	`int8_t out_port;`
	`char out_signal;`
	
	`void output() {`
	`  if (out_signal) {`
	`  out_port |= MASK1;`
	`  } else {`
	`  out_port &= ~MASK1;`
	`  }`
	`  write_byte(0x01, 0x02, out_port);`
	`}`

Таблица 3 – Трансляция переменных, отображаемых на входные и выходные порты.

Пояснение к таблице: MASK1 – битовая маска, у которой все биты, кроме младшего, нулевые. В общем случае, при отображении n-ого бита порта на переменную, используется битовая маска, у которой все биты, кроме n-1-ого, нулевые.

2.2.5 Временные интервалы

Временные интервалы могут встречаться в выражениях и в определении времени активизации процесса. Временной интервал трансформируется в число, соответствующее значению временного интервала в миллисекундах.

2.2.6 Аннотации

Аннотации используются для задания дополнительной информации, которая нужна кодогенератору или предметно-ориентированному модулю. Далее в работе будет описан пример использования аннотаций для целей генерации кода, зависимого от аппаратной платформы.

2.3 Подходы к созданию предметно-ориентированных языков

Предметно ориентированный язык программирования (DSL) – язык программирования, предназначенный для решения круга задач определенной предметной области. Предметно-ориентированные языки облегчают разработку и поддержку программных систем с особой спецификой, позволяя абстрагироваться от средств языков общего назначения [7].

В настоящее время существует большое множество инструментов, облегчающих процесс разработки предметно-ориентированных языков. В частности, к таким средствам относятся *парсер-генераторы*. Парсер-генераторы принимают на вход описание грамматики входного языка в нужной нотации и порождают программу на определенном языке (парсер), которая может осуществлять лексический и синтаксический анализ исходного кода.

Также среди средств создания предметно-ориентированных языков выделяют полноценные программные среды для разработки DSL. Такие инструменты позволяют с помощью малых усилий получать полную инфраструктуру поддержки языка для IDE.

Таким образом, можно выделить два основных подхода создания инструментальных средств для поддержки предметно-ориентированных языков [8]:

1. «Ручной» подход. Предполагает использование парсер-генераторов (ANTLR, Flex & Bison) и самостоятельное проектирование архитектуры системы, написание и интеграцию средств семантического анализа, функционального редактора.

   - *Плюсы подхода:* полный контроль над исходным кодом и архитектурой системы.
   - *Минусы подхода:* крайне неудобен для построения сложных программных систем, таких как IDE.

2. Использование фреймворков для разработки DSL (Xtext, JetBrains MPS)

   - *Минусы подхода:* сильная зависимость от выбранных технологий, возможность столкнуться с различными ограничениями фреймворков.
   - *Плюсы подхода:* продуманная архитектура, наличие вспомогательных средств и библиотек, упрощающих реализацию различных компонент языка (семантический анализ, кодогенератор, редактор кода).

2.4 Обзор и сравнительный анализ средств парсер-генерации и создания предметно-ориентированных языков

Для рассмотрения были выбраны GNU Bison, ANTLR4, JetBrains MPS и Xtext как наиболее известные и используемые средства парсер-генерации и разработки DSL.

1. GNU Bison – программа для генерации синтаксических анализаторов [9]. В качестве нотации для описания грамматик Bison использует БНФ, а код, необходимый для создания синтаксического дерева нужно явно прописывать в файле с грамматикой. Такой подход усложняет грамматики, делает их плохо читаемыми и увеличивает затраты на поиск ошибок в грамматике и в парсере. Синтаксический анализатор использует LALR(1) и GLR-алгоритмы анализа. Bison был разработан для Unix-подобных операционных систем, однако существует портирование и под Windows. Парсер-генератор может строить парсеры на языках C, C++ и Java. Для работы парсер-генератора нужен сторонний лексический анализатор (обычно используется flex). Стоит отметить, что данное средство является довольно устаревшим.
2. ANTLR4 – современный парсер-генератор [10]. В качестве нотации грамматик ANTLR использует РБНФ, что облегчает описание грамматик. В качестве алгоритма синтаксического анализа используется ALL(*)-алгоритм, разработанный создателем ANTLR на основе LL(*), эффективность которого была исследована и показана в [11]. Средство является кроссплатформенным, поскольку написано на языке Java. ANTLR поддерживает генерацию парсера на множестве языков, к которым относятся C++, Java, Python, JavaScript. Парсер, сгенерированный ANTLR, строит синтаксическое дерево – древовидную структуру, для которой предоставляются методы обхода в глубину. Существует ряд инструментальных средств, облегчающих разработку при использовании ANTLR: интегрированная среда разработки ANTLR Studio, а также плагины для Eclipse и IntelliJ IDEA.

3. Xtext [12] – среда разработки текстовых языков на основе Eclipse-платформы, которая использует средства парсер-генерации. Xtext предлагает гибкий язык описания грамматик, основанный на РБНФ. По описанию грамматики Xtext строит парсер, множество классов для хранения AST, а также полную инфраструктуру среды разработки для целевого языка: синтаксически-ориентированный редактор с подсветкой ошибок, поддержкой автодополнений, семантического анализа, кодогенерации. В качестве парсер-генератора Xtext использует ANTLR3, при этом расширяя его возможности с точки зрения описания грамматики и построения AST. При этом стоит отметить, что ANTLR3, в отличие от ANTLR4, использует обычный LL(*)-алгоритм. Огромным преимуществом Xtext для поставленной задачи является то, что все языки, разработанные с помощью этого средства, поддерживают Language Server Protocol (LSP). Таким образом, результаты данной работы могут быть использованы при интеграции языка Reflex в различные IDE, которые поддерживают этот протокол [13]. Перечень таких IDE представлен в электронном ресурсе [14], к наиболее известным из них относятся Sublime, VSCode, IntelliJ IDEA.

4. JetBrains MPS [15] – среда разработки проекционных языков на основе платформы Intellij IDEA. Так же как и Xtext, JetBrains MPS строит полную инфраструктуру среды разработки языка. Вместо использования способа, когда для построения синтаксического дерева используется парсер, JetBrains MPS предлагает подход, при котором синтаксическое дерево редактируется напрямую с помощью проекционного редактора [16]. Для хранения программ, созданных в таком редакторе, используется xml-подобный формат данных. Такой подход дает преимущества в виде добавления в язык различных графических элементов (таблиц, диаграмм, форм и т.д.) и вставок на других языках. Этот фреймворк может быть использован и для текстовых языков, однако это приводит к ряду издержек – ввиду особенностей формата хранения, программы на языках, созданных с помощью JetBrains MPS, могут отображаться и редактироваться довольно узким набором средств. Созданные с помощью JetBrains MPS инструментальные средства могут быть интегрированы в среду разработки IntelliJ IDEA в качестве плагина.

В таблицах 4 и 5 приведено краткое обобщение результатов сравнения средств по критериям, наиболее важным для поставленной задачи.

	Алгоритм парсера	Кроссплатформенность	Построение AST	Удобство описания грамматик	Наличие IDE
Xtext	LL(*)	+	+	+	+
ANTLR4	ALL(*)	+	±	+	+
Bison & flex	LALR(1), GLR	±	-	-	±

Таблица 4 – Сравнение средств, использующих парсер-генерацию

	Кроссплатформенность	Генерация синтаксически-ориентированного редактора	Средства семантического анализа	Поддержка кодогенерации	Переносимость созданных программ	Интеграция созданных средств в сторонние IDE
Xtext	+	+	+	+	+	+
JetBrains MPS	+	+	+	+	-	-

Таблица 5 – Сравнение фреймворков для создания DSL.

По результатам сравнительного анализа средств парсер-генерации и создания DSL были сделаны следующие выводы:

- С точки зрения парсер-генерации, Xtext не имеет серьезных недостатков по сравнению с Bison и ANTLR4, а по некоторым критериям превосходит их;
- JetBrains MPS является довольно сложным для освоения средством, а также обладает серьезными недостатками с точки зрения переносимости программ и интеграции созданных инструментальных средств в какие-либо IDE, кроме IntelliJ;
- Xtext создает полноценную инфраструктуру поддержки языка для множества сред разработки. Таким образом, работа по построению интерактивного редактора кода для IDE может быть выполнена автоматически, при этом обеспечивается малая зависимость созданных инструментальных средств от технологий Eclipse.

В соответствии с выводами, в качестве вспомогательного средства был выбран Xtext.

2.5 Архитектура разрабатываемой программной системы

2.5.1 Компоненты разрабатываемой программной системы и их взаимодействие

Архитектура разрабатываемой системы по большей части соответствует стандартной архитектуре системы, которую предлагает Xtext. Таким образом, можно выделить следующие части системы:

- **Редактор**, в котором пользователь набирает текст программы.
- **Парсер**, который получает текст, набранный в редакторе, производит синтаксический разбор и создает синтаксическое дерево. В случае успешного построения, синтаксическое дерево передается синтаксическому анализатору и делегирующему кодогенератору. В случае неудачи, информация об ошибках передается редактору кода.
- **Компоненты семантического анализа**. Компоненты семантического анализа производят ряд проверок над синтаксическим деревом. В случае обнаружения семантических ошибок, средства передают информацию о них редактору кода. В случае отсутствия ошибок синтаксическое дерево передается делегирующему кодогенератору.
- **Делегирующий кодогенератор**. Получает на вход синтаксическое дерево и передает его известным ему проблемно-ориентированным модулям.
- **Кодогенератор для языка Си** является предметно-ориентированным модулем, с которым взаимодействует делегирующий кодогенератор**.** Он принимает синтаксическое дерево, производит обход его узлов и генерирует набор текстовых файлов с определенной структурой. В общем случае, предметно-ориентированным модулем может являться любой модуль, использующий абстрактное синтаксическое дерево для генерации каких-либо артефактов.

Эти части и их взаимодействие изображены на рисунке 1. Зеленым цветом отмечены части системы, которые создаются автоматически по описанию грамматики с помощью Xtext, синим – те части, которые предстоит реализовать.

Инфраструктура взаимодействия редактора, парсера, компонент семантического анализа и делегирующего кодогенератора также автоматически обеспечивается Xtext. Таким образом, для организации архитектуры необходимо разработать механизм взаимодействия делегирующего кодогенератора с предметно-ориентированными модулями.

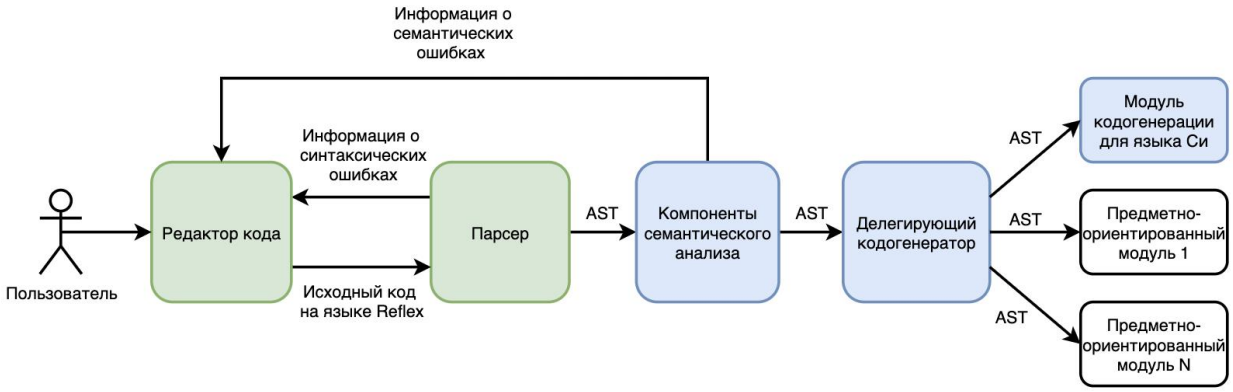


Рисунок 1 – Компоненты системы и их взаимодействие

2.5.2 Структура модулей системы

Все проекты, создаваемые с помощью Xtext, имеют общую структуру, которая генерируется при создании проекта [23]. Для организации модульности в Xtext используются средства Eclipse Platform. Так, проект состоит из слабо связанных модулей – Eclipse-плагинов . В таблице 6 преставлены плагины, из которых состоит программная система, а также описано их содержимое.

Название плагина	Компоненты, входящие в плагин	Примечание
ru.iaie.reflex	Парсер	+
	Классы для хранения AST	+
	Средства семантического анализа	*
	Делегирующий кодогенератор	*
	Набор утилитных классов со вспомогательными методами для работы с AST	*
ru.iaie.reflex.generators.r2c	Генератор кода для языка Си	*
ru.iaie.reflex.ide	Компоненты поддержки языка для IDE, независимые от Eclipse-платформы (поддержка Language Server Protocol)	+
ru.iaie.reflex.ui	Компоненты пользовательского интерфейса для Eclipse IDE	+

Таблица 6 – Плагины, из которых состоит система. Знаком «*» отмечены компоненты плагинов, которые реализуются в рамках ВКР, знаком «+» - те, которые Xtext создает автоматически.

В ходе проектирования было установлено, что Xtext предполагает наличие единственного кодогенератора, который по умолчанию находится том же модуле, что парсер и семантический анализатор. Это нарушает из требование к создаваемой программной системе, при котором является модуль парсера должен быть независим от модуля

кодогенерации, а также является препятствием для расширения системы (добавления множества кодогенераторов в проект). Решением является создание делегирующего кодогенератора, с которым непосредственно будет взаимодействовать Xtext, а также вынесение создаваемых кодогенераторов в отдельные Eclipse-плагины. При этом, взаимодействие между делегирующим генератором и внешними кодогенераторами обеспечивается посредством механизма Eclipse extensions. [17]

Так, плагин, содержащий делегирующий кодогенератор, декларирует extension point для интерфейса `IReflexGenerator`. Плагины кодогенерации объявляют extension (расширение) в виде класса, реализующего этого интерфейс. Таким образом, с помощью средств Eclipse Platform делегирующий кодогенератор может осуществлять поиск объектов классов, реализующих интерфейс `IReflexGenerator`, а затем вызывать у этих объектов метод, отвечающий за кодогенерацию.

Стоит отметить, что предложенный способ ориентирован на IDE, разрабатываемую с помощью технологий Eclipse. Инфраструктура взаимодействия делегирующего кодогенератора с проблемно-ориентированными модулями может меняться при интеграции разработанных средств в другие IDE.

3 Реализация инструментальных средств для языка Reflex и их апробация на тестовой задаче

Реализация инструментальных средств производилась на языке Xtend – диалекте языка Java. Язык Xtend расширяет Java «синтаксическим сахаром», который позволяет более элегантно и быстро описывать программы. При программировании на Xtend можно использовать java-библиотеки. Xtend транслируется в Java [18].

3.1 Внутреннее представление программы

Для создания внутреннего представления программ и работы с ним Xtext использует средства Eclipse Modelling Framework (EMF). [19]

Парсер, сгенерированный Xtext, создает абстрактное синтаксическое дерево в виде EMF-модели (далее будем использовать термин *семантическая модель программы* и подразумевать, что это понятие в Xtext эквивалентно понятию AST).

По описанию грамматики Xtext строит описание *метамодели* в терминах EMF. Метамодель представляет собой некоторую схему данных, описывающую структуру EMF-модели, которая будет использована для хранения AST. Правила грамматики становятся

сущностями метамодели. Сущности могут иметь атрибуты: ссылки на другие сущности (связи один-к-одному, один-ко-многим) или значения примитивных типов. На основе метамодели языка генерируется иерархия Java-классов. При этом сущности метамодели становятся классами, а их атрибуты – полями этих классов. Все сгенерированные классы реализуют интерфейс `EObject` – интерфейс элемента EMF-модели. [20]

Такой подход, в силу свойств EMF-моделей, позволяет обращаться с AST с одной стороны, как с графом (например, выполнять над каждым узлом операции получения родительского и дочерних узлов, выполнять операцию поиска узла в дереве, удовлетворяющего некоторому предикату), а с другой стороны – как с объектной моделью (в нашем случае, рассматривать корневой элемент дерева как объект, соответствующий программе, у которого можно получить список объектов-процессов, у которых можно получить список объектов-состояний, переменных и так далее) (листинг 8).

```
Program program = ast.findNodeOfType(Program);/* Получение корневого
элемента AST - программы */
Process process = program.getProcesses().get(0) /* Получение начального
процесса программы */
State state = program.getStates(0) /* Получение начального состояния
процесса*/
```

Листинг 8 – Работа с AST (псевдокод)

Поскольку описание семантической модели программы на уровне грамматики не позволяет гибко определить необходимое множество операций над моделью, в ходе работы был реализован набор вспомогательных методов для работы с AST программ на языке Reflex.

3.2 Описание грамматики языка Reflex в нотации Xtext

Xtext имеет специальный DSL для описания LL-грамматик, который основан на расширенной форме Бэкуса-Наура. Отличительной особенностью этого языка является тесная связь с семантической моделью программы. [21]

Таким образом, в языке описания грамматик Xtext присутствуют следующие средства:

1. Операторы +=, ?= и = для описания атрибутов сущностей синтаксического дерева. Слева от операторов указывается название атрибута, а справа - название правила грамматики;
2. Возможность описания ссылок (cross-references) в грамматике. Справа от оператора присваивания также можно указать ссылку – имя правила грамматики в квадратных скобках «[]». Это будет означать, что на месте ссылки парсер будет распознавать последовательность символов, которая в ссылаемом правиле

помечена специальным атрибутом `name`. При этом в синтаксическое дерево на этапе разрешения ссылок (linking) будет попадать ссылка на конкретный объект, атрибут `name` которого соответствует распознанной парсером последовательности символов.

Эти средства использовались при описании грамматики языка Reflex. В листинге 9 представлено описание правила, соответствующего программе на Reflex. Здесь использовался оператор «=» для сохранения в синтаксическое дерево атрибутов, соответствующих имени программы и периоду активизации. Для сохранения списка перечислений, констант, сигнатур Си-функций, глобальных переменных и описаний процессов использовался оператор «+=».

Пример использования ссылок в грамматике Reflex представлен в листинге 10. Идентификатор процесса объявляется атрибутом правила `Process` с именем `name`. В правиле `StartProcStat` атрибутом `process` является не просто идентификатор, а ссылка на `Process`. Таким образом, парсером будет распознаваться последовательность **start process** <*идентификатор процесса*>, при этом в синтаксическое дерево попадет объект, соответствующий этому процессу с указанным идентификатором (в случае, если такой процесс был объявлен в Reflex-программе).

```
Program:
"program" name=ID "{"
    ticks=Tact
    (consts+=Const |
    enums+=Enum |
    functions+=Function |
    globalVars+=GlobalVariable |
    registers+=Register)*
          processes+=Process*
    "}";
```

Листинг 9 – Правило Program в нотации Xtext.

```
Process:
"process" name=ID "{"
    variables+=ProcessVariable*
    states+=State*
"}";

StartProcStat:
    "start" "process"
    process=[Process] ";";
```

Листинг 10 – Использование механизма разрешения ссылок в правиле запуска процесса.

С полным описанием грамматики языка Reflex в нотации Xtext можно ознакомиться в приложении А.

3.3 Реализация модуля кодогенерации

Стандартным интерфейсом кодогенератора в Xtext является `org.eclipse.xtext.generator.IGenerator2`, который содержит следующие методы:

- `beforeGenerate(org.eclipse.emf.ecore.resource.Resource input, IFileSystemAccess2 fsa, IGeneratorContext context)` – метод, который вызывается перед осуществлением генерации кода;
- `doGenerate(org.eclipse.emf.ecore.resource.Resource input, IFileSystemAccess2 fsa, IGeneratorContext context)` – метод, реализующий логику генерации кода;
- `afterGenerate(org.eclipse.emf.ecore.resource.Resource input, IFileSystemAccess2 fsa, IGeneratorContext context)` – метод, который вызывается после генерации кода.

Важными аргументами этих методов являются `input` и `fsa`. Аргумент `input` содержит AST-дерево в виде EMF-модели, аргумент `fsa` используется для доступа к файловой системе, аргумент `context` может содержать дополнительную информацию, которая необходима кодогенератору.

Этот интерфейс было решено взять за основу интерфейса кодогенераторов для языка Reflex `IReflexGenerator`.

3.3.1 Структура генерируемых файлов

В результате кодогенерации создаются три папки – generated, lib, usr. Наглядно структура файлов представлена на рисунке 2.

Рассмотрим детально назначение и содержимое каждой из папок:

1. В папке lib содержатся вспомогательные файлы для кодогенерации:

   1.1. *r_cnst.h* – константы и определения типов, общие для всех транслируемых программ;

   1.2. *r_lib.h, r_lib.c* – объявления и определения вспомогательных функции, которые могут быть используются во всех генерируемых программах;

   1.3. *platform.h* – заголовочный файл с объявлением функций, реализация которых зависит от от аппаратной платформы, в которой будет выполняться программа. Эти функции имеют следующие сигнатуры и назначение:

• `uint32_t get_time()` – получение текущего значения таймера устройства (в миллисекундах);

• `void init_io()` – инициализация портов ввода-вывода;

• `void init_time()` – инициализация таймера;

• `int8_t read_byte(int, int), int16_t read_word(int, int)` – чтение значения с восьмибитного/шестнадцатибитного порта. Аргументами этих функций являются числа, определяющие адрес порта на устройстве;

• `int write_byte(int, int, int8_t), int write_word(int, int, int16_t)` – запись значения на восьмибитный/шестнадцатибитный порт. Первые два аргумента определяют адрес порта, третьим аргументом является значение, которое надо записать.

2. Папка *usr* содержит объявления и определения пользовательских Си-функций, которые используются в Reflex-программе. Объявления задаются пользователем в заготовочном файле *usr.h*, определения – в *usr.c*.

3. В папке *generated* содержатся файлы, непосредственно созданные генератором

3.1. *cnst.h* – заголовочный файл, в который попадают определения констант и перечислений программы;

3.2. *ext.h* – заголовочный файл, который используется для компоновки всех заголовочных файлов, необходимых для работы программы;

3.3. *gvar.h* – объявления всех переменных программы;

3.4. *xvar.h* – заголовочный файл, который используется для доступа к переменным, описанным в gvar.h;

3.5. *io.h*, *io.c* – объявления и определения функций `input()` и `output()`, о которых говорилось ранее в разделе «Трансформационная семантика для языка Си»;

3.6. *proc.h*, *proc.c* – файлы с объявлениями и реализациями функций-процессов;

3.7. *main.c* – реализация функции main программы. Функция main включает в себя бесконечный цикл гиперпроцесса;

3.8. *platform.c* – реализация функций из заголовочного файла *platform.h*

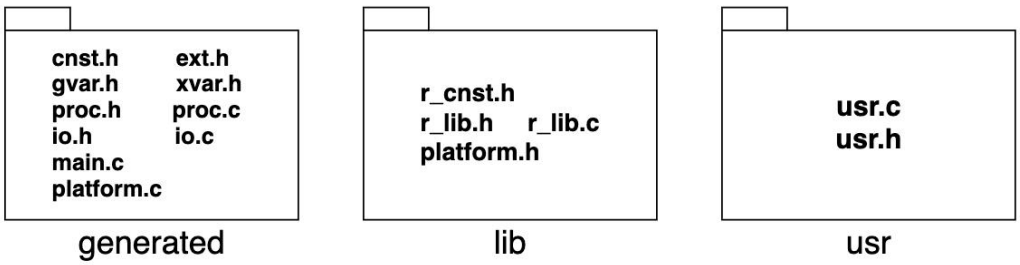


Рисунок 2 – Структура генерируемых файлов

Содержимое файлов *r_cnst.h*, *r_lib.h*, *r_lib.c* из папки *lib* было заимствовано у старого транслятора с некоторыми модификациями.

3.3.2 Структура модуля кодогенерации и алгоритм кодогенерации

На рисунке 3 изображена диаграмма классов модуля кодогенерации. Более детализованная диаграмма классов представлена в приложении В. Рассмотрим алгоритм генерации кода в терминах классов и их методов.

Класс `R2CReflexGenerator` реализует интерфейс `IReflexGenerator`. Он содержит в качестве своего поля объект класса `R2CFileGenerator`.

Конструктор класса `R2CFileGenerator` в качестве аргументов принимает корневой элемент синтаксического дерева и объект типа `IFileSystemAccess2`, который используется для доступа к файловой системе. Класс предоставляет набор методов для генерации структуры программы:

- Метод `prepareForGeneration()` производит копирование папки lib в место назначения, генерирует файл ext.h, а также производит поиск файлов с именем usr.h и usr.c в той же директории, где находится файл с исходным кодом на языке Reflex. В случае нахождения таких файлов, они копируются в папку usr, иначе - создаются в папке usr.
- Метод `generateVariableDefinitions()` генерирует файлы gvar.h и xvar.h. Для генерации объявлений переменных и переменных, которые хранят значения портов используются вспомогательные классы `VariableGenerationHelper` и `PortGenerationHelper`.
- Метод `generateConstantDefinitions()` генерирует файл cnst.h. Для генерации констант и перечислений используется класс `ConstantGenerationHelper`.
- Метод `generateProcessDefinitions()` осуществляет генерацию файла proc.h.

• Метод `generateProcessImplementations()` производит генерацию файла proc.c. Для генерации функций-процессов используется класс `ProcessGenerator`.

• Метод `generateIO()` генерирует файлы io.h и io.c. Для генерации кода считывания/записи значений с портов и отображения этих значений на переменные программы используется класс `PortGenerationHelper`.

• Метод `generatePlatformImplementations()` отвечает за генерацию файла platform.c.

• `generateMain()` создает файл main.c

Все методы класса `R2CFileGenerator` вызываются внутри функции `doGenerate` класса `R2CReflexGenerator`. Порядок вызовов этих методов не важен.

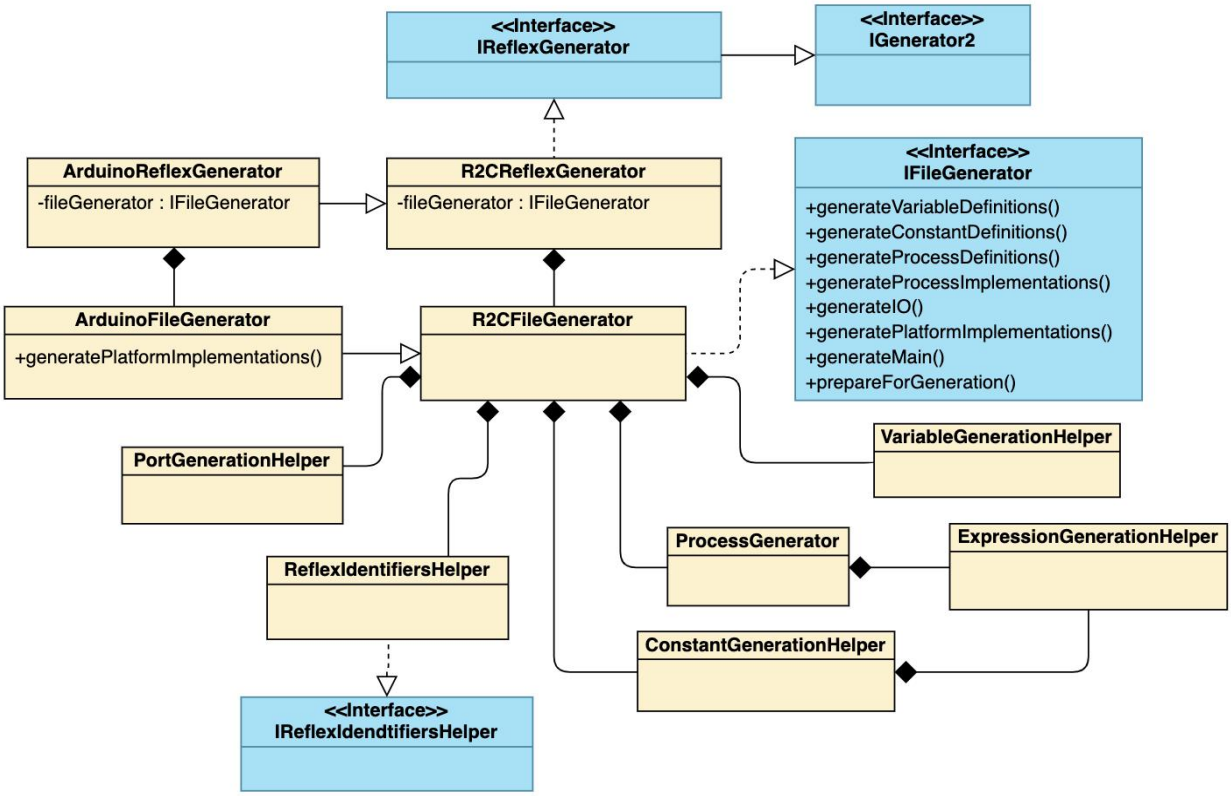


Рисунок 3 – Диаграмма классов модуля кодогенерации

При трансляции имен процессов, констант, перечислений, переменных (в объявлениях и выражениях), портов к ним добавляются префиксы. Такой подход был использован по нескольким причинам:

• Избавление от ситуаций, при которых происходит коллизия служебных имен, используемых в сгенерированном коде с идентификаторами пользовательского кода;

- Согласно правилам трансляции, все переменные языка Reflex трансформируются в глобальные переменные языка Си. Добавление префиксов к именам позволяет обеспечить корректную генерацию кода в случае, если в разных процессах объявлены локальные переменные с одинаковым именем.

Для генерации идентификаторов используется класс `ReflexIdentifiersHelper`. Этот класс отображает идентификаторы программы на языке Reflex в идентификаторы, которые будут использованы в сгенерированном коде.

Класс предоставляет методы для генерации идентификаторов для каждого типа объектов синтаксического дерева, имена которых используются в генерируемом коде (к таким объектам относятся процесс, порт, перечисление, член перечисления, константа, глобальная переменная, переменная процесса).

При вызове такого метода сначала проверяется, присутствует ли имя объекта из AST (оригинальное имя) в хэш-таблице, которая соответствует методу (например, в таблице имен констант). Если идентификатор присутствует, то значение берется из хэш-таблицы, иначе – генерируется имя *префикс_объекта+оригинальное_имя_объекта*. Значения вызовов методов кэшируются с помощью хэш-таблиц (для каждого метода используется своя хэш-таблица).

Для генерации кода используются «template expressions» – способ создания строк в языке Xtend с помощью шаблонов, которые позволяют делать произвольные подстановки в строку, а также управлять созданием строки с помощью конструкций цикла и условного перехода [25]. Пример использования этого приема при реализации кодогенератора представлен в Листинге 11.

```
def generateProcessFunction(Process process) {
    return '''
    void process«process.index»() {
        switch (processes[«process.index»].cur_state) {
        «FOR state : process.states»
            case «state.index»: {
                  «FOR stat : state.statements»
                      «translateStatement(state, stat)»
                  «ENDFOR»
                  «IF state.hasTimeoutReaction»
                    «translateTimeoutReaction(state)»
                  «ENDIF»
                break;
            }
          «ENDFOR»
        }
     }
    '''
}
```

Листинг 11 – Использование template expressions на примере метода генерации функций-процессов (Язык Xtend)

3.3.3 Генерация кода под платформу Arduino

Выбранная иерархия классов модуля кодогенерации и структура генерируемых файлов, при которой аппаратно-зависимый код выделен в отдельный файл и его реализация скрыта за интерфейсными методами, существенно упрощает создание кодогенераторов, ориентированных на различные аппаратные платформы. В качестве тестовой платформы в данной работе была выбрана аппаратная платформа Arduino на базе AVR-микроконтроллера ATmega168.

Кодогенератор был реализован в виде класса `ArduinoReflexGenerator`, который является наследником класса `R2CReflexGenerator` (см. диаграмму классов на рисунке 3). Главное изменение заключалось в реализации класса `ArduinoFileGenerator`, который наследуется от класса `R2CFileGenerator`. `ArduinoFileGenerator` переопределяет метод `generatePlatformImplementations()`, генерируя содержимое файла platform.c, которое соответствует тестовой платформе. Таким образом, общая структура файлов и алгоритм их генерации не меняется, происходит только изменение логики генерации файла platform.c.

Для управления генерацией файла platform.c предлагается использование аннотаций. Программа может быть помечена набором аннотаций, значение которых должно представлять собой код на языке Си. Значения аннотаций подставляются в определенные структурные части файла platform.c. Названия и назначение аннотаций, которые могут быть использованы для управления кодогенерацией под платформу Arduino, представлены в таблице 7.

Название аннотации	Назначение
`ArduinoPLC.timer_init`	Значение аннотации подставляется в тело функции init_time
`ArduinoPLC.timer_function`	Значение аннотации подставляется в тело функции get_time
`ArduinoPLC.io_init`	Значение аннотации подставляется в тело функции init_io
`ArduinoPLC.read_byte`, `ArduinoPLC.read_word`, `ArduinoPLC.write_byte`, `ArduinoPLC.write_word`	Значения аннотаций подставляются в тело функций чтения/записи значений с портов, название которых соответствуют названию аннотации. Для доступа к аргументам функций, внутри подстановки могут быть использованы имена `addr1, addr2` (аргументы, определяющие работу системный адрес порта на устройстве) и `data` (аргумент, определяющий записываемое значение)

`ArduinoPLC.insert_global`	Значение аннотации подставляется вне определений функций. Таким образом, в файл можно подставлять директивы #include, а также объявления глобальных переменных и определения функций

Таблица 7 – Названия и назначение аннотаций, распознаваемых модулем кодогенерации под платформу Arduino.

Если для какой-либо структурной части файла platform.c подстановка не определена с помощью аннотации, то используется стандартная реализация. Реализация функций работы со временем была заимствована из проекта IndustrialC [26]. Стандартная реализация функции инициализации портов ввода-вывода `init_io()` для микроконтроллера ATmega168 предполагает инициализацию всех битов портов B и D как выходных и всех битов порта C как входных. Функции чтения и записи значений портов в стандартной реализации идентифицируют порт в соответствии со следующим отображением: порту B соответствует число 0, порту C - число 1, порту D - число 2. Для доступа к регистрам записи, чтения и инициализации портов используется библиотека `avr/io.h`.

3.4 Семантический анализ кода на языке Reflex средствами Xtext

Xtext создает инфраструктуру, которая значительно упрощает реализацию семантических проверок для языка. К механизмам семантического анализа в Xtext можно отнести validation и linking.

Так, для каждого языка можно определить реализацию валидатора. Валидатор представляет собой класс, наследуемый от абстрактного класса `AbstractDeclarativeValidator` пакета `org.eclipse.xtext.validation` с набором методов, помеченных специальной аннотацией `@Check` [24]**.** Эти методы должны иметь один аргумент типа, который могут иметь узлы синтаксического дерева. После генерации синтаксического дерева Xtext осуществляет обход его узлов и вызывает соответствующие им методы валидатора. Методы валидатора могут генерировать ошибки и предупреждения с помощью статических методов `error()` и `warning()`, реализацию которых предоставляет родительский класс.

В листинге 12 представлен пример валидатора с функцией проверки наличия следующего состояния для выражения **set next state.** Проверка осуществляется функцией `checkForNextState` с аргументом типа `SetNextStateStat`**.**

Сначала находятся узлы дерева, соответствующие состоянию и процессу, в котором было объявлено такое выражение. Находится индекс этого состояния в процессе. Если индекс является последним, то с помощью функции `error()` генерируется сообщение об ошибке. При этом сообщение об ошибке будет локализованным: в редакторе слева от строки с ошибкой появится *маркер*, сигнализирующий об ошибке, также сообщение об ошибке будет доступно при наведении на строку и попадет в специальный виджет IDE Problems (рисунок 4).

```
class ReflexValidator extends AbstractDeclarativeValidator {
    @Check
    func checkForNextState(SetNextStateStat statement) {
          State state = statement.getParentNodeOfType(State)
          Process process = statement.getParentNodeOfType(Process)
          int stateIndex = process.states.indexOf(state)
          if (stateIndex == process.states.size - 1) {
                error("Invalid state transition:
                           no next state in the process")
          }
    }
}
```

Листинг 12 – Валидатор с методом проверки наличия следующего состояния для выражения перехода. (псевдокод)

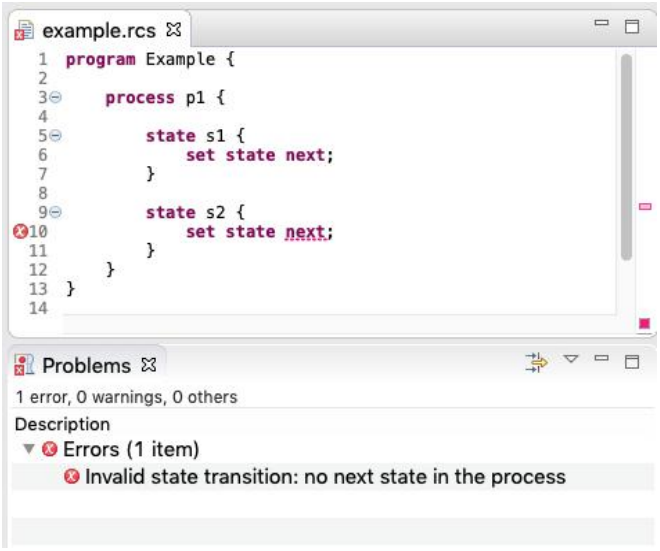


Рисунок 4. Отображение ошибки семантического анализа

Другое средство семантического анализа в Xtext - это linking (линковка, разрешение ссылок). Применение этого средства - реализация проверок наличия декларации используемых имен и связывание используемых имен с объектами, на которые ссылаются эти имена, на уровне AST. Работа этого механизма обеспечивается реализацией следующих вещей [20]:

1. Описание ссылок на уровне грамматики. В разделе «Описание грамматики языка Reflex средствами Xtext» было отмечено, что в Xtext существует такая возможность и ее применение было продемонстрировано на примере в листинге 10.
2. Описание семантики разрешения ссылок посредством scoping API. Scoping API представляет собой класс, реализующий интерфейс `IScopeProvider` с методом `getScope`, который используется для подбора множеств объектов, являющихся кандидатами на разрешение ссылки. Реализация метода, предлагаемая Xtext по умолчанию, использует простейшие алгоритмы выбора множеств. Таким образом, в некоторых случаях приходится изменять реализацию этого метода.

Рассмотрим применение механизма на примере реализации правила проверки наличия в процессе состояния, на которое ссылается выражение **set state** <*имя состояния*>.

Наличие ссылки указывается в грамматике. Помимо этого, переопределяется реализация метода `getScope` (листинг 13), поскольку стандартная реализация метода возвращает множество состояний всех процессов в программе, в то время как областью поиска должны быть только состояния процесса, в котором объявлено выражение перехода.

Метод `getScope` имеет два аргумента. В аргументе `context` передается объект синтаксического дерева, в котором возникает ссылка, в аргументе `ref` – тип ссылки. По

двум этим аргументам можно однозначно разрешить множество значений. В примере из листинга обрабатывается случай, при котором `context` – это объект типа `SetStateStat`, а аргумент `ref` указывает на то, что разрешается ссылка на объект типа `State` для выражения **set state.** В этом случае для объекта `context`, который соответствует выражению перехода, производится поиск родительского элемента-процесса и из метода возвращается множество объектов-состояний этого процесса.

```
class ReflexScopeProvider extends AbstractScopeProvider {
    @Override
    func IScope getScope(EObject context, EReference ref)  {
        if (context is SetStateStat && ref == STATE_IN_SET_STATE){
             Process process = context.getParentNodeOfType(Process);
             return Scopes.scopeFor(process.states);
        }
        return super.getScope(context, ref)
     }
}
```

Листинг 13 – Разрешение ссылки на состояние в выражении перехода (псевдокод)

В приложении С описан полный набор семантических проверок, которые были реализованы для языка Reflex, а также средства Xtext, которые были задействованы при их реализации.

3.5 Редактор кода

Созданная программная система представляет собой приложение, наследующее основную функциональность Eclipse IDE, с редактором кода, ориентированным на язык Reflex. Редактор поддерживает подсветку синтаксиса, а также отображение синтаксических и семантических ошибок. В приложении присутствует область навигации по проектам. Проект представляет собой папку, в которой содержится файл с исходным кодом на языке Reflex. Файлы с исходным кодом должны иметь расширение *.rcs. При сохранении результатов редактирования (посредством нажатия в редакторе сочетания клавиш Ctrl+S) приложение осуществляет кодогенерацию, результаты которой попадают в папку src-gen проекта.

Внешний вид редактора кода представлен на рисунке 5.

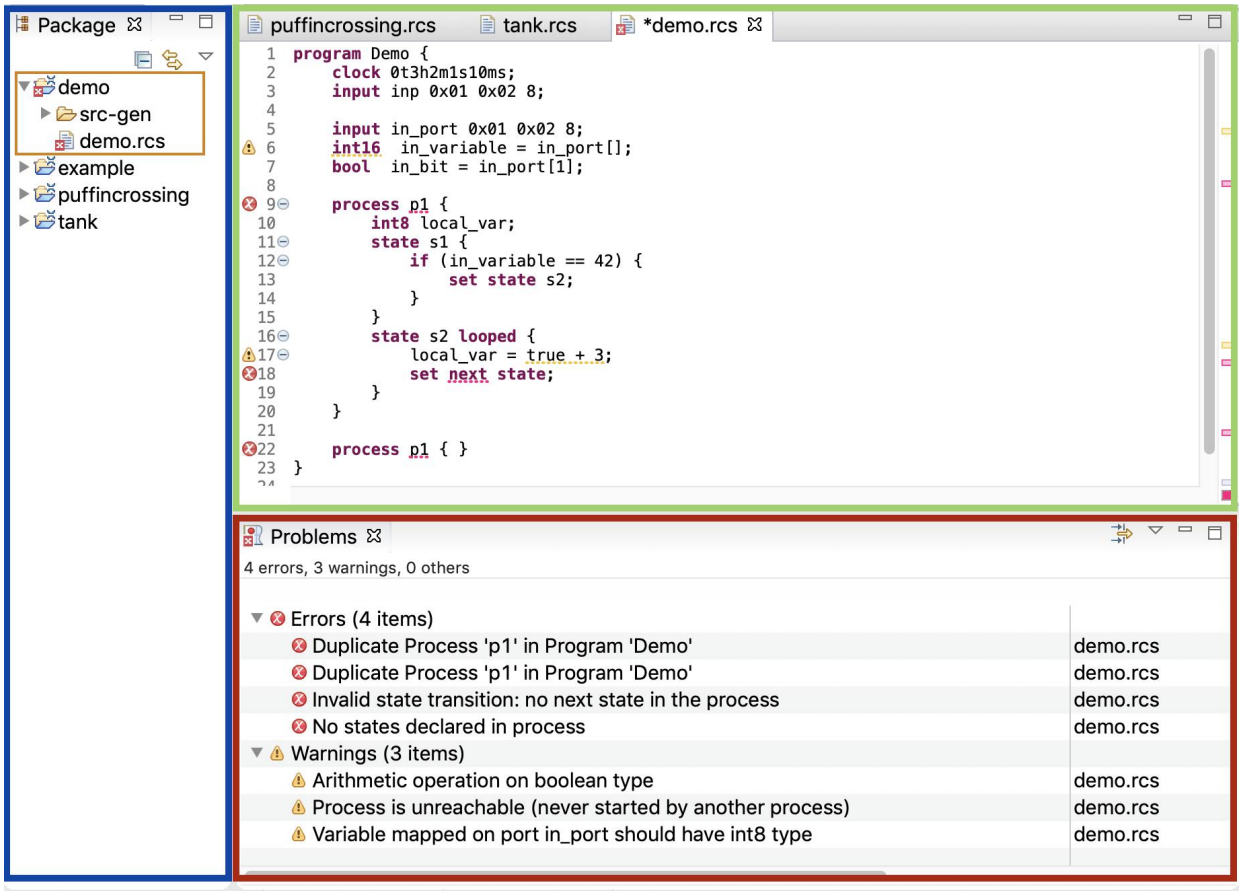


Рисунок 5 – Внешний вид редактора кода. Синим цветом выделена область навигации по проектам, желтым - структура проекта, зеленым - область редактора кода, красным - область отображения ошибок.

3.6 Апробация разработанных средств

Распознавание парсером семантических конструкций, а также контроль срабатывания семантических проверок тестировались посредством набора примеров программ на языке Reflex в редакторе.

Проверка работоспособности сгенерированного кода проводилась на тестовой задаче «Сушилка для рук». «Сушилка для рук» представляет собой систему с однобитным входным сигналом (сигнал с датчика присутствия рук) и однобитным выходным сигналом (сигнал управления подачей воздуха). Требования задачи:

- при поднесении рук к датчику должен начинать подаваться воздух;
- во время нахождения рук рядом с датчиком воздух должен подаваться непрерывно;
- после того, как руки были убраны из-под датчика, подача воздуха должна прекратиться в течение определенного интервала времени (для наглядности демонстрации при тестировании был выбран интервал в 2 секунды)

Полный листинг программы на языке Reflex, соответствующий тестовой задаче, представлен в приложении D.

Апробация осуществлялась посредством компиляции сгенерированного кода компилятором avr-gcc, а затем загрузки .hex-файла в микроконтроллерную плату, которая подсоединена к схеме со светодиодом и кнопкой (кнопка имитирует входной сигнал, светодиод имитирует выходной сигнал). Апробация проводилась на плате Arduino Nano на базе микроконтроллера ATmega168. При апробации на тестовой задаче наблюдалось выполнение требований, которые были описаны выше.

ЗАКЛЮЧЕНИЕ

По итогам работы были получены следующие результаты:

1. с помощью фреймворка Xtext реализован модуль, осуществляющий парсинг, построение абстрактного синтаксического дерева и семантический анализ исходных кодов на языке Reflex;
2. реализован модуль генерации кода для языка Си, ориентированный на платформу Arduino;
3. произведена апробация созданных инструментальных средств на тестовой задаче.

Реализованные средства были использованы при построении Web-IDE для языка Reflex [27].

В дальнейшем планируется добавление возможности описания программы в нескольких файлах, расширение синтаксиса аннотаций и исследование их применимости для реализации модулей анализа программ на Reflex, разработка универсальной архитектуры модуля кодогенерации.

Результаты работы были представлены на Международной научной студенческой конференции 2020 в секции «Информационные технологии» в подсекции «Программная архитектура и системное программирование» [28] и отмечены дипломом III степени.

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

Бастрыкина Алена Алексеевна
ФИО студента

_____
Подпись студента

« ____ »_____2020г.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Зюбин, В. Е. Язык «Рефлекс» – диалект Си для программируемых логических контроллеров // Шестая международная научно-практическая конференция «Средства и системы автоматизации» CSAF. – Т. 6.

2. Зюбин, В. Е. Процесс-ориентированное программирование: Учеб. пособие / В. Е. Зюбин – Новосиб. гос. ун-т. – 2011. – 194 с.

3. Rozov A. S. Process-oriented programming language for MCU-based automation / Rozov A. S., Zyubin V. E. // 2013 International Siberian Conference on Control and Communications (SIBCON). – IEEE, 2013. – С. 1-4.

4. Зюбин, В. Е. Базовый модуль, управляющий установкой для выращивания монокристаллов кремния / В. Е. Зюбин, В. Н. Котов, Н. В. Котов и др. // Датчики и системы. – 2004. – №. 12. – С. 17-22.

5. Горнев И. А. Создание среды разработки для языка Reflex: диплом. работа бак. / Горнев И. А. – Новосиб. гос. ун-т. – 2019. – 35 с.

6. Zyubin V. E. Reflex Language: a Practical Notation for Cyber-Physical Systems / Zyubin V. E., Liakh T. V., Rozov A. S. // Системная информатика. – 2018. – №. 12. – С. 85-104.

7. Dmitriev S. Language oriented programming: The next programming paradigm // JetBrains onBoard. – 2004. – Т. 1. – №. 2. – С. 1-13.

8. Federico Tomasetti. The complete guide to (external) Domain Specific Languages: What tools can we use to build Domain Specific Languages? [Электронный ресурс] / – Режим доступа: https://tomassetti.me/domain-specific-languages/#tools

9. Levine J. Flex & Bison: Text Processing Tools. // – " O'Reilly Media, Inc.", 2009.

10. ANTLR4 Documentation. [Электронный ресурс] / – Режим доступа: https://github.com/antlr/antlr4/blob/master/doc/index.md

11. Adaptive LL (*) parsing: the power of dynamic analysis / Parr T., Harwell S., Fisher K. // ACM SIGPLAN Notices. – 2014. – Т. 49. – №. 10. – С. 579-598.

12. Bettini L. An Eclipse-based IDE for Featherweight Java implemented in Xtext // Eclipse IT. – Eclipse Italian community, 2010. – С. 14-28.

13. Bünder H. Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages // Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019). – 2019. – С. 131-142.

14. A community-driven source of knowledge for Language Server Protocol implementations [Электронный ресурс] / – Режим доступа: https://langserver.org/

15. JetBrains MPS, Meta Programming System [Электронный ресурс] / – Режим доступа: https://www.jetbrains.com/mps/

16. Ботов, Д. С. Обзор современных средств создания и поддержки предметно-ориентированных языков программирования // Вестник Южно-Уральского государственного университета. Серия: Компьютерные технологии, управление, радиоэлектроника. – 2013. – Т. 13. – №. 1.

17. Notes on the Eclipse Plug-in Architecture. The Eclipse Plug-in Model. Extension [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html#2

18. Xtend - Java Interoperability [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/xtend/documentation/201_types.html

19. EMF Documentation [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/modeling/emf/docs/

20. Xtext - Integration with EMF [Электронный ресурс] / – Режим доступа: eclipse.org/Xtext/documentation/308_emf_integration.html

21. Xtext - Grammar Language [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html

22. Xtext - Language Implementation. Linking [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#linking

23. Lorenzo Bettini. Implementing Domain-Specific Languages with Xtext and Xtend / Lorenzo Bettini. – Packt Publishing, 2016. – С. 23.

24. Behrens H. et al. Xtext user guide / – Режим доступа: https://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf – 2008. – С. 45.

25. Xtend - Template Expressions [Электронный ресурс] / – Режим доступа: https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates

26. IndustrialC. A process-oriented language for embedded microcontroller programming [Электронный ресурс] / – Режим доступа: https://github.com/deadproger/IndustrialC

27. Марченко, К. В. Использование Eclipse Theia для создания интегрированной среды разработки программ на процесс-ориентированном языке Reflex / К. В. Марченко // Материалы 58-й Международной научной студенческой конференции МНСК-2020: Информационные технологии / Новосиб. гос. ун-т. Новосибирск, 2020. с. 146.

28. Бастрыкина, А. А. Рефакторинг транслятора языка Reflex на основе автоматической парсер-генерации / А. А. Бастрыкина // Материалы 58-й Международной научной студенческой конференции МНСК-2020: Информационные технологии / Новосиб. гос. ун-т. Новосибирск, 2020. с. 138.

ПРИЛОЖЕНИЕ А. Грамматика Reflex в нотации Xtext

```
grammar ru.iaie.reflex.Reflex with org.eclipse.xtext.common.Terminals

generate reflex "http://www.iaie.ru/reflex/Reflex"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Program:
      ("[" annotations+=Annotation "]")*
      "program" name=ID "{"
      clock=ClockDefinition
      (consts+=Const |
      enums+=Enum |
      functions+=Function |
      globalVars+=GlobalVariable |
      ports+=Port |
      processes+=Process)*
      "}";

ClockDefinition:
      "clock" (intValue=INTEGER | timeValue=TIME) ";";

Process:
      ("[" annotations+=Annotation "]")*
      "process" name=ID "{"
      ((imports+=ImportedVariableList | variables+=ProcessVariable) ";")*
      states+=State*
      "}";

State:
      ("[" annotations+=Annotation "]")*
      "state" name=ID (looped?="looped")? "{"
      stateFunction=StatementSequence
      (timeoutFunction=TimeoutFunction)?
      "}";

Annotation:
      key=AnnotationKey ":" value=STRING | key=AnnotationKey;

AnnotationKey:
      ID "." ID | ID;

ImportedVariableList:
      "shared" (variables+=[ProcessVariable] ","
variables+=[ProcessVariable])*) "from" "process" process=[Process];

ProcessVariable:
      (PhysicalVariable | ProgramVariable) (shared?="shared")?;

GlobalVariable:
      (PhysicalVariable | ProgramVariable) ";";

PhysicalVariable:
      type=Type name=ID "=" mapping=PortMapping;
```

```
PortMapping:
      port=[Port] "[" (bit=INTEGER)? "]";

ProgramVariable:
      type=Type name=ID;

TimeoutFunction:
      "timeout" (TimeAmountOrRef | "(" TimeAmountOrRef ")")
body=Statement;

fragment TimeAmountOrRef:
      time=TIME | intTime=INTEGER | ref=[IdReference];

Function:
      returnType=Type name=ID "(" argTypes+=Type ("," argTypes+=Type)*
")" ";";

Port:
      type=PortType name=ID addr1=INTEGER addr2=INTEGER size=INTEGER ";";

enum PortType:
      INPUT='input' | OUTPUT='output';

Const:
      "const" type=Type name=ID "=" value=Expression ";";

Enum:
      "enum" identifier=ID "{" enumMembers+=EnumMember (','
enumMembers+=EnumMember)* "}";

EnumMember:
      name=ID ("=" value=Expression)?;

      // Statements
Statement:
      {Statement} ";" | CompoundStatement |
      StartProcStat | StopProcStat | ErrorStat | RestartStat | ResetStat
      | SetStateStat | IfElseStat | SwitchStat | Expression ";";

StatementSequence:
      {StatementSequence} statements+=Statement*;

CompoundStatement:
      {CompoundStatement} "{" statements+=Statement* "}";

IfElseStat:
      "if" "(" cond=Expression ")"
      then=Statement
      (=> "else" else=Statement)?;

SwitchStat:
      "switch" "(" expr=Expression ")" "{" options+=CaseStat*
defaultOption=DefaultStat? "}";

CaseStat:
      "case" option=Expression ":" "{" SwitchOptionStatSequence "}";

DefaultStat:
```

```
      "default" ":" "{" SwitchOptionStatSequence "}";

fragment SwitchOptionStatSequence:
      statements+=Statement* hasBreak?=BreakStat?;

BreakStat:
      "break" ";";

StartProcStat:
      "start" "process" process=[Process] ";";

StopProcStat:
      {StopProcStat} "stop" ("process" (process=[Process]))? ";";

ErrorStat:
      {ErrorStat} "error" ("process" (process=[Process]))? ";";

RestartStat:
      {RestartStat} "restart" ";";

ResetStat:
      {ResetStat} "reset" "timer" ";";

SetStateStat:
      {SetStateStat} "set" ((next?="next" "state") | ("state"
state=[State])) ";";

IdReference:
      PhysicalVariable | ProgramVariable | EnumMember | Const;

      // Expressions
InfixOp:
      op=InfixPostfixOp ref=[IdReference];

PostfixOp:
      ref=[IdReference] op=InfixPostfixOp;

FunctionCall:
      function=[Function] "(" (args+=Expression ("," args+=Expression)*)?
")";

CheckStateExpression:
      "process" process=[Process] "in" "state" qualfier=StateQualifier;

enum StateQualifier:
      ACTIVE="active" | INACTIVE="inactive" | STOP="stop" | ERROR="error";

PrimaryExpression:
      reference=[IdReference] | {PrimaryExpression} integer=INTEGER |
{PrimaryExpression} floating=FLOAT |
      {PrimaryExpression} bool=BOOL_LITERAL | {PrimaryExpression}
time=TIME | "(" nestedExpr=Expression ")";

UnaryExpression:
      PrimaryExpression |
      FunctionCall |
      PostfixOp |
      InfixOp |
```

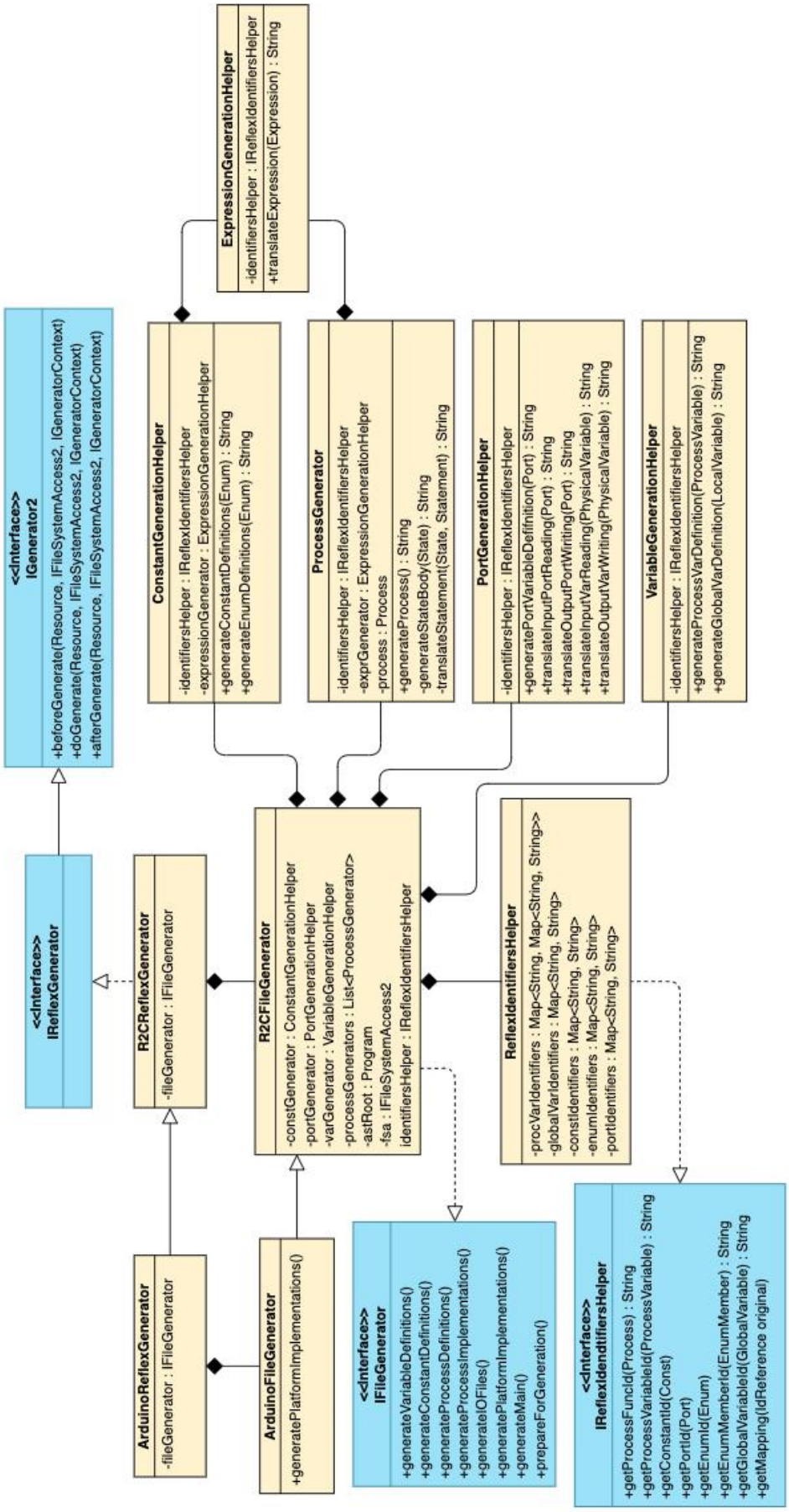
```
      unaryOp=UnaryOp right=CastExpression;

CastExpression:
      UnaryExpression |
      "(" type=Type ")" right=CastExpression;

MultiplicativeExpression:
      CastExpression ({MultiplicativeExpression.left=current}
mulOp=MultiplicativeOp right=CastExpression)*;

AdditiveExpression:
      MultiplicativeExpression ({AdditiveExpression.left=current}
addOp=AdditiveOp right=AdditiveExpression)*;

ShiftExpression:
      AdditiveExpression ({ShiftExpression.left=current} shiftOp=ShiftOp
right=ShiftExpression)*;

CompareExpression:
      CheckStateExpression | ShiftExpression
({CompareExpression.left=current} cmpOp=CompareOp
right=CompareExpression)*;

EqualityExpression:
      CompareExpression ({EqualityExpression.left=current}
eqCmpOp=CompareEqOp right=EqualityExpression)*;

BitAndExpression:
      EqualityExpression ({BitAndExpression.left=current} BIT_AND
right=BitAndExpression)*;

BitXorExpression:
      BitAndExpression ({BitXorExpression.left=current} BIT_XOR
right=BitXorExpression)*;

BitOrExpression:
      BitXorExpression ({BitOrExpression.left=current} BIT_OR
right=BitOrExpression)*;

LogicalAndExpression:
      BitOrExpression ({LogicalAndExpression.left=current} LOGICAL_AND
right=LogicalAndExpression)*;

LogicalOrExpression:
      LogicalAndExpression ({LogicalOrExpression.left=current} LOGICAL_OR
right=LogicalOrExpression)*;

AssignmentExpression:
      (assignVar=[IdReference] assignOp=AssignOperator)?
expr=LogicalOrExpression;

Expression:
      AssignmentExpression;

enum InfixPostfixOp:
      INC="++" | DEC="-";

enum AssignOperator:
```

```
      ASSIGN="=" | MUL='*=' | DIV="/=" | MOD="+=" | SUB="-=" | CIN="<<="
| COUT=">>=" | BIT_AND="&=" | BIT_XOR="^=" |
      BIT_OR="|=";

enum UnaryOp:
      PLUS="+" | MINUS="-" | BIT_NOT="~" | LOGICAL_NOT="!";

enum CompareOp:
      LESS="<" | GREATER=">" | LESS_EQ="=<" | GREATER_EQ=">=";

enum CompareEqOp:
      EQ="==" | NOT_EQ="!=";

enum ShiftOp:
      LEFT_SHIFT=">>" | RIGHT_SHIFT="<<";

enum AdditiveOp:
      PLUS="+" | MINUS="-";

enum MultiplicativeOp:
      MUL="*" | DIV="/" | MOD="%";

terminal LOGICAL_OR:
      "||";

terminal LOGICAL_AND:
      "&&";

terminal BIT_OR:
      "|";

terminal BIT_XOR:
      "^";

terminal BIT_AND:
      "&";

// Types
enum Type:
      VOID_C_TYPE="void" | FLOAT="float" | DOUBLE="double" | INT8="int8"
| INT8_U="uint8" | INT16="int16" |
      INT16_U="uint16" | INT32="int32" | INT32_U="uint32" | INT64="int64"
| INT64_U="uint64" | BOOL="bool" | TIME="time";

// Literals
terminal INTEGER:
      SIGN? (HEX | OCTAL | DECIMAL) (LONG (UNSIGNED)? | UNSIGNED
(LONG)? )?;

terminal FLOAT:
      DEC_FLOAT | HEX_FLOAT;

terminal fragment DEC_FLOAT:
      DEC_SEQUENCE? '.' DEC_SEQUENCE (EXPONENT SIGN DEC_SEQUENCE)? (LONG
| FLOAT_SUFFIX)?;

terminal fragment HEX_FLOAT:
```

```
      HEX_SEQUENCE? '.' HEX_SEQUENCE (BIN_EXPONENT SIGN DEC_SEQUENCE)?
(LONG | FLOAT_SUFFIX)?;

terminal fragment DEC_SEQUENCE:
      ('0'..'9')+;

terminal fragment HEX_SEQUENCE:
      ('0'..'9' | 'a'..'f' | 'A'..'F')+;

terminal fragment BIN_EXPONENT:
      ('p' | 'P');

terminal fragment EXPONENT:
      'e' | 'E';

terminal fragment SIGN:
      '+' | '-';

terminal fragment DECIMAL:
      "0" | ('1'..'9') ('0'..'9')*;

terminal fragment OCTAL:
      '0' ('0'..'7')+;

terminal fragment HEX:
      HEX_PREFIX HEX_SEQUENCE;

terminal fragment HEX_PREFIX:
      '0' ('x' | 'X');

terminal fragment LONG:
      "L" | "l";

terminal fragment FLOAT_SUFFIX:
      "F" | "f";

terminal fragment UNSIGNED:
      "U" | "u";

terminal TIME:
      ("0t" | "0T") (DECIMAL DAY)? (DECIMAL HOUR)? (DECIMAL MINUTE)?
(DECIMAL SECOND)? (DECIMAL MILLISECOND)?;

terminal fragment DAY:
      "D" | "d";

terminal fragment HOUR:
      "H" | "h";

terminal fragment MINUTE:
      "M" | "m";

terminal fragment SECOND:
      "S" | "s";

terminal fragment MILLISECOND:
      "MS" | "ms";
```

```
terminal BOOL_LITERAL returns ecore::EBooleanObject:
      "true" | "false";
```

ПРИЛОЖЕНИЕ В. Диаграмма классов модуля кодогенерации



ПРИЛОЖЕНИЕ С. Набор семантических проверок для языка Reflex

Проверка	Тип реакции на нарушение правила (ошибка/предупреждение)	Механизм Xtext, используемый для реализации проверки
Проверка наличия следующего состояния в процессе для выражения set next state	Ошибка	validation
Проверка наличия состояния в текущем процессе для выражения set state <Sname>	Ошибка	linking
Контроль доступности переменных, используемых в выражениях (по области видимости)	Ошибка	linking
Контроль отсутствия замкнутых состояний (состояний, не содержащих оператора перехода в другое состояние, при этом не помеченных явно модификатором looped)	Ошибка	validation
Контроль отсутствия недостижимых состояний (состояний, для которых нет операции перехода из других состояний процесса)	Предупреждение	validation
Контроль отсутствия попыток присвоить значение переменной, отображаемой на входной порт	Предупреждение	validation
Контроль отсутствия присвоения значений переменным, которые отображаются на выходные порты	Предупреждение	validation
Контроль инициализации переменных, используемых в качестве интервала в выражении timeout	Ошибка	validation
Проверка отсутствия недостижимых процессов (таких, которые не запускаются с помощью оператора start из других процессов)	Предупреждение	validation

Контроль отсутствия присваивания значений константам и членам перечислений	Ошибка	validation
Контроль инициализации констант и членов перечислений только константными выражениями	Ошибка	validation
Контроль наличия процесса в программе, используемого в выражениях start | stop | error <Pname>, process <Pname> in state <active | inactive | error | stop>	Ошибка	linking
Контроль типов в выражениях	Предупреждение	validation
Контроль отсутствия дублирования имен состояний, процессов, переменных, констант, членов перечислений	Ошибка	validation
Контроль наличия в процессе разделяемой переменной для конструкции shared <Vname> from <Pname>	Ошибка	linking
Контроль использования объявленных переменных	Предупреждение	validation
Контроль наличия выражений в теле оператора **timeout**	Ошибка	validation
Контроль наличия процессов в программе и состояний в процессе	Ошибка	validation

ПРИЛОЖЕНИЕ D. Листинг программы на языке Reflex для задачи «Сушилка для рук»

```
program Dryer {
    clock 0t10ms;
    input inp 0x00 0x00 8;
    output out 0x00 0x01 8;

    const bool ON = true;
    const bool OFF = false;

    const time TIMEOUT = 0t2s;

    process Dryer {
        bool hands_under_dryer = inp[1];
        bool dryer_control = out[1];
        state Wait {
            if (hands_under_dryer) {
                dryer_control = ON;
                set state Work;
            }
        }
        state Work {
            if (hands_under_dryer) reset timer;
            timeout (TIMEOUT) {
                dryer_control = OFF;
                set state Wait;
            }
        }
    }
}
```