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
Бурдэ Сергея Петровича

Тема работы:

РАЗРАБОТКА МОДУЛЯ ГЕНЕРАЦИИ ИСПОЛНЯЕМОГО КОДА ДЛЯ МИКРОКОНТРОЛЛЕРА ATMEGA ПО AST REFLEX-ПРОГРАММЫ

«К защите допущен»
Заведующий кафедрой,
ФИТ НГУ, 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. СИ ФИТ НГУ
Ануреев И.С. /……..
(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 технологий

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)
«02» ноября 2023г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Бурдэ Сергею Петровичу, группы 20204
(фамилия, имя, отчество, номер группы)

Тема работы: Разработка модуля генерации исполняемого кода для микроконтроллера ATmega по AST Reflex-программы
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 02.11.2023 г. №0409
Срок сдачи студентом готовой работы «20» мая 2024 г.
Исходные данные (или цель работы): разработка модуля генерации исполняемого кода для микроконтроллера ATmega по AST Reflex-программы
Структурные части работы: анализ предметной области, расширение языка Reflex, имплементация парсера и модуля кодогенерации, исследование на модельных примерах.
Консультанты по разделам ВКР отсутствуют

Руководитель ВКР
Заведующий кафедрой КТ
ФИТ НГУ, д.т.н., доцент
Зюбин В.Е./……..
(ФИО) / (подпись)
«02» ноября 2023г.

Задание принял к исполнению

Бурдэ С. П. /……..
(ФИО студента) / (подпись)
«02» ноября 2023г.

Соруководитель ВКР
к.ф-м.н.,
доцент каф. СИ ФИТ НГУ
Ануреев И.С. /……..
(ФИО) / (подпись)
«02» ноября 2023г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Встраиваемые системы используются во многих областях, включая промышленную автоматизацию, бытовую электронику, автомобильную промышленность и медицинские устройства. Встраиваемые системы, которые зачастую реализуются на базе микроконтроллеров ATmega, требуют высокоэффективных алгоритмов управления, которые должны работать в условиях ограниченных вычислительных ресурсов. Поэтому возникает необходимость в создании инструментов, которые значительно упростят процесс разработки надежного и эффективного программного обеспечения для микроконтроллеров.

Языки программирования Reflex и Industrial-C, разрабатываемые и исследуемые в Институте Автоматики и Электрометрии СО РАН, уже показали свою эффективность в этой области [1, 2].

В процессе разработки программ на данном языке возникло множество предложений по его расширению и корректировке синтаксиса.

Цель работы – коррекция синтаксиса и модуля кодогенерации языка Reflex для целей программирования распределенных систем на базе микроконтроллеров ATmega.

Для достижения этой цели были поставлены следующие **задачи**:

1) Анализ специфики программирования встраиваемых систем.
2) Анализ специфики существующих процесс-ориентированных языков Reflex, Industrial-C.
3) Формулировка требований к разрабатываемым средствам.
4) Коррекция языка Reflex.
5) Имплементация парсера и модуля кодогенерации.
6) Тестирование разработанных инструментальных средств и исследование проведенных коррекций на модельных примерах.

Научная новизна:

Синтаксис процесс-ориентированного языка программирования Reflex дополнен новыми концептами и конструкциями, расширяющими функционал языка.

Практическая ценность:

Снижение трудоемкости и стоимости разработки программ для микроконтроллеров ATmega. Использование специализированного языка программирования Reflex повышает читаемость кода программ на базе микроконтроллеров и снижает требования к квалификации разработчика.

Структура работы:

Работа состоит из введения, трех глав и заключения.

В первой главе анализируется специфика предметной области и формулируются требования к разрабатываемым языковым и инструментальным средствам. Во второй главе описывается синтаксис и семантика новых конструкций языка Reflex. В третьей главе описывается имплементация парсера и модуля кодогенерации для языка Reflex, описываются результаты исследования языка на модельных примерах.

Глава 1. Анализ предметной области

1.1. Специфика предметной области

Программирование встраиваемых систем и микроконтроллеров имеет уникальные особенности, обусловленные спецификой этих платформ. В отличие от классических вычислительных систем, микроконтроллеры обладают ограниченными вычислительными ресурсами, такими как низкие тактовые частоты, малый объем ОЗУ и ПЗУ, и отсутствие активного охлаждения. Важными аспектами являются работа с аппаратными ресурсами, включая регистры и прерывания, что позволяет микроконтроллерам оперативно реагировать на внешние события в реальном времени, а также взаимодействие с встроенными периферийными устройствами, такими как порты ввода-вывода, таймеры и аналого-цифровые преобразователи (АЦП).

Кроме того, программирование встраиваемых систем предполагает прямой доступ к аппаратным ресурсам, таким как регистры и встроенные периферийные устройства.

Помимо этого, алгоритмы управления в задачах автоматизации обладают рядом специфических свойств, отличающих их от традиционных вычислительных задач [3]. Среди этих свойств:

1) Открытость: наличие внешней среды и необходимость постоянного взаимодействия с ней. Система управления должна реагировать на изменения во внешней среде, формируя управляющие воздействия на основе получаемой информации. Внешняя среда включает в себя объект управления и оператора системы, а взаимодействие с ней осуществляется через датчики и исполнительные устройства.
2) Логический параллелизм: поскольку события в физической среде происходят независимо друг от друга, управление должно осуществляться параллельно. В микроконтроллерах, где отсутствуют возможности для физического параллелизма, параллельная обработка

событий реализуется программно, через многозадачность и логический параллелизм.

3) Синхронизм – синхронизация исполнения управляющего алгоритма с физическими процессами на объекте управления.
4) Событийность

Все эти особенности и свойства определяют требования к методикам и языкам программирования, используемых при разработке встраиваемых систем на базе микроконтроллеров.

Процесс-ориентированный подход обеспечивает выполнение перечисленных требований. В основе этого подхода лежит понятие гиперпроцесса – набора взаимодействующих процессов. Каждый процесс представлен конечным автоматом, который расширен операциями для работы с данными и межпроцессного взаимодействия [4].

Данный подход разрабатывается и тестируется в Институте Автоматики и Электрометрии СО РАН (ИАиЭ СО РАН) на примере языков программирования Reflex и Industrial-C. Использование языков для решения реальных задач показало, что процесс-ориентированный подход упрощает разработку алгоритмов автоматизации [5,6,7].

1.2. Анализ специфики языков Reflex и Industrial-C

Языки программирования Reflex и Industrial-C были разработаны в Институте автоматики и электрометрии СО РАН для решения задач автоматизации на базе программируемых логических контроллеров и микроконтроллеров. Оба языка представляют собой процесс-ориентированные диалекты языка Си, обеспечивающие поддержку специализированных конструкций для управления сложными системами в реальном времени. Для обоих языков реализованы трансляторы в язык Си.

1.2.1. Анализ недостатков

В процессе программирования на данных языках накопилось множество предложений по расширению и коррекции синтаксиса языка Reflex:

1) Введение в язык концепта узлов, позволяющих задать распределенную микроконтроллерную топологию вычислительной системы.
2) Введение в существующий синтаксис конструкций для упрощенного задания состояний, так называемых легковесных состояний.
3) Введение в язык конструкций, позволяющих взаимодействовать с векторами, регистрами, битами микроконтроллеров, как это реализовано в Industrial-C.
4) Коррекция спецификации отображения переменных на физические порты ввода-вывода (далее такие переменные будем называть физическими). Имеющийся способ не предоставляет возможность конфигурации отдельных битов порта как входных или выходных. Помимо этого, декларация физических переменных должна предусматривать возможность задания прямого и отложенного отображения. Концепция прямого отображения была заимствована из стандарта IEC 61131-3, описанная в стандарте, как “directly represented variables” [8].

1.2.2. Сравнительный анализ

В таблице 1 представлены обобщенные результаты сравнения Reflex и Industrial-C.

Таблица 1 – сравнение языков Reflex и Industrial-C

	Reflex	Industrial-C
Средство реализации транслятора в язык Си	Eclipse/Xtext	flex & bison
Процесс-ориентированный	+	+
Диалект языка Си	+	+
Кроссплатформенность	+	-

Работа с периферией	+-	+
Работа с векторами, регистрами, битами	-	+
Возможность задания узлов распределенной топологии	-	-
Инструменты упрощенного задания состояний (легковесные состояния)	-	-

По результатам сравнительного анализа можно сделать следующие выводы:

- Оба языка реализуют процесс-ориентированный подход, являются диалектами языка Си.
- Industrial-C реализован с помощью Flex & Bison [9], которые являются более устаревшими технологиями по сравнению с Eclipse Xtext, на котором реализован Reflex. Flex и Bison требуют ручного написания значительного объема кода, что может быть трудоемким и подверженным ошибкам процессом. Xtext значительно упрощает создание языков и инструментов для них, предоставляя высокоуровневые абстракции и автоматизируя многие аспекты, такие как создание синтаксического и семантического анализа, IDE-поддержки и многое другое [10].
- Reflex обладает кроссплатформенностью, в отличае от Industrial-C, который разработан конкретно с микроконтроллерами AVR.
- Reflex слабо поддерживает работу с периферией (порты ввода-вывода), и не поддерживает работу с векторами прерываний, регистрами, что является критически важным при программировании встраиваемых систем. Industrial-C, напротив, хорошо работает с векторами и регистрами, но не является кроссплатформенным.
- Оба языка не реализуют работу с узлами и легковесными состояниями.

1.3. Список требований к разрабатываемым средствам

На основе проведенного анализа был сформулирован список требований к коррекции языка и разрабатываемым средствам:

1) Язык должен предусматривать возможность задания распределенной микроконтроллерной топологии вычислительной системы с указанием узлов и привязки процессов к этим узлам.
2) Для каждого специфицированного узла модуль кодогенерации должен формировать соответствующую Си-проекцию.
3) Язык должен предусматривать возможность задания легковесных состояний
4) Язык должен предусматривать возможность взаимодействия с регистрами, битами и векторами для полноценной работы с периферией микроконтроллера и обработки прерываний.
5) Декларация переменных, отображаемых на порты ввода-вывода, должна предусматривать возможность задания прямого и отложенного отображения, регистров чтения/записи/конфигурации, полного или частичного отображения

1.4. Основные выводы главы

В ходе анализа предметной области были выявлены ключевые особенности программирования встраиваемых систем на базе микроконтроллеров ATmega, такие, как работа в условиях ограниченных ресурсов, необходимость детерминированного поведения в реальном времени и тесное взаимодействие с аппаратными компонентами. Сравнительный анализ языков Reflex и Industrial-C показал, что каждый из них имеет свои сильные и слабые стороны: Reflex обладает кроссплатформенностью, но ограничен в поддержке периферии, тогда как Industrial-C хорошо работает с аппаратурой, но не является кроссплатформенным.

Для улучшения этих языков предложены расширения синтаксиса, включая концепты узлов и легковесных состояний, что может значительно упростить процесс разработки. Основные требования к новым инструментам включают поддержку легковесных состояний, распределенной топологии вычислительных систем. На основе этих требований будут введены новые конструкции, расширяющие язык Reflex, описанные в следующей главе.

Глава 2. Расширение языка Reflex.

Для улучшения языка Reflex и его адаптации к программированию на микроконтроллерах ATmega были внесены следующие ключевые изменения в синтаксис.

2.1. Узлы

Одним из основных расширений языка Reflex стало введение концепта узлов. Узлы позволяют задавать распределенную топологию вычислительной системы, в которой каждый узел представляет собой отдельный микроконтроллер. Объявление узла начинается с ключевого слова node и названия узла <имя_узла>, после чего описываются его переменные, константы, перечисления, обработчики прерываний и период активизации процессов. Такая структура позволяет четко определить границы каждого узла и управлять его взаимодействием с другими узлами системы.

Вместе с конструкцией объявления узлов была введена конструкция спецификации принадлежности процессов к узлу. Это осуществляется при указании после имени процесса конструкции :: node <имя_узла> (листинг 1).

```
program Example{
    clock 0t10ms;
    node Node1 {
        clock 0t10ms;
        const int8 CAN_ID = 0x01;
    }
    node Node2{
        clock 0t10ms;
        const int8 CAN_ID= 0x02;
    }
    active process p1 :: node Node1{
    }
    process p2 :: node Node2{
    }
}
```

Листинг 1 – объявление и использование узлов

Семантика данных конструкций включает в себя следующие аспекты:

- Объявление узла подразумевает создание в результате трансляции отдельного файла кода для каждого узла (<имя_узла>.c), в котором будут объявлены все переменные, константы и прочие элементы, описанные для этого узла.
- Привязка процесса к узлу означает, что код процесса будет сгенерирован в соответствующем файле узла, к которому он привязан (таблица 2).

Таблица 2 – конструкции узлов

Конструкция	Семантика	Кодогенерация

node <nodeID> {
переменные, константы…
}
	объявление узла переопределение clock, констант, переменных, enum	Си-файл “nodeID.c” с переменными, константами и т.д. из этого узла

process Test2 ::
node <nodeID> {
переменные процесса,
состояния
}
	привязка процесса к узлу	Код процесса будет сгенерирован в файле “nodeID.c”

2.2. Легковесные состояния

Для улучшения языка Reflex и его адаптации к программированию на микроконтроллерах ATmega была введена концепция легковесных состояний. К ним относятся конструкции slice, transition, transition on timeout.

Легковесные состояния значительно упрощают и оптимизируют написание управляющих алгоритмов, снижая сложность кода и повышая его читаемость. Эти упрощенные структуры позволяют избежать создания лишних состояний и присвоения им уникальных имен, обеспечивая более компактное представление логики процесса.

Использование slice, transition и transition on timeout допускается только в состояниях, наряду с внешними инструкциями (top-level statement), то есть использование конструкций легковесных состояний во вложенных блочных операциях приводит к ошибке.

В таблице 3 представлен синтаксис и семантика конструкций легковесных состояний, а также определен эквивалент кода на Reflex, который реализуется соответствующими конструкциями.

Таблица 3 – конструкции легковесных состояний (slice, transition, transition on timeout)

Конструкция	Семантика	Эквивалент

slice;	введение неявного состояния и передача управления	set next state;
		}
		state slice_1 {

transition (<condition>);	введение неявного состояния; по семантике похоже на переход (transition) в сетях Петри	set next state;
		}
		state transition_1{
		if (<condition>)
		set next state;
		}
		state transition_2{

transition (<condition>) on timeout(<time>) {<reaction>};	к семантике transition добавляется механизм сторожевого таймера (watchdog) и реакцию по тайм-ауту	set next state;}
		state transition_1{
		if (<condition>)
		set next state
		else timeout(<time>)
		{<reaction>}
		}
		state transition_2{

2.3. Регистры, биты, векторы

В Reflex добавлена возможность декларации регистров микроконтроллера, битов этих регистров и векторов прерываний. Объявление регистра начинается с ключевого слова register, бита — с bit, вектора — с vector.

Идентификаторы в этих объявлениях должны совпадать с соответствующими идентификаторами ресурсов, описанными в подключаемом заголовочном файле конкретного микроконтроллера, для которого описываются эти ресурсы. Этот файл подключается в platform.h – header file, в котором описан весь платформозависимый функционал.

Объявленные регистры и биты могут использоваться для отображения физических переменных на порты микроконтроллера, а векторы — для регистрации обработчиков прерываний. Кроме того, регистры могут быть использованы в выражениях как для чтения, так и для записи. Биты, в свою очередь, могут использоваться только для чтения.

Объявления ресурсов микроконтроллера могут быть сгруппированы в структуре import <nameID>.

Все эти объявления являются служебными: сами объявления не транслируются в Си, а в Си транслируются только их идентификаторы, используемые в других конструкциях языка. Использование идентификаторов регистров, битов или векторов без их объявления приводит к ошибке.

2.4. Физические переменные

От конструкции port было решено отказаться, так как она не предусматривала возможность конфигурации отдельных битов порта как входных или выходных. Вместо этого весь порт целиком определялся как входной или выходной.

Для взаимодействия с портами ввода-вывода используются физические переменные, описание которых выглядит следующим образом:
<тип отображения> <тип переменной> <имя переменной> as <отображение порта>

Отображение порта (PortMapping) — это новая конструкция языка, начинающаяся с ключевых слов input или output, которые определяют, происходит ли отображение на физический вход или выход. Далее в круглых скобках через запятую указываются регистры чтения, записи, конфигурации, а также может быть указан бит порта. Указание регистра конфигурации является опциональным; указание регистра чтения обязательно для отображений на физический вход (input), а указание регистра записи обязательно для отображений на физический выход (output). В качестве регистра может быть указана целочисленная константа (адрес регистра в микроконтроллере). Если бит не указан, физическая переменная отображается на весь порт. В листинге 2 описана простая программа на Reflex с использованием физических переменных.

```
program SmartLamp {
	clock 0t10ms;
	const bool ON = true;
	const bool OFF = false;
	const time TIMEOUT = 0t30s;
	const uint8 pinb = 0x25;
	register PORTB;
	register DDRB;
	bit PB0;
	bit PB1;

	direct bool movement_detected as input(
		read=pinb,
		config=DDRB,
		bit=PB0
	);
	direct bool light_control as output(
		write=PORTB,
		config=DDRB,
		bit=PB1
	);
	active process Light_switching{
		state Wait {
			if (movement_detected) {
				light_control = ON;
				set state Work;
			}
		}
		state Work {
			if (movement_detected) reset timer;
			timeout (TIMEOUT) {
				light_control = movement_detected;
				set state Wait;
			}
		}
	}
}
```

Листинг 2 – программа «Умная лампа» на Reflex с использованием физических переменных

2.4.1. Прямое и отложенное отображение

Физическая переменная может быть прямо отображена на регистры ввода-вывода микроконтроллера, если указать тип отображения direct. Прямое отображение означает, что при генерации Си кода, обращения к

соответствующей переменной будет заменено на обращение к регистру, объявление переменной транслироваться не будет.

Отложенное (не прямое) отображение выбирается по умолчанию, если не указывать тип отображения, либо если указать тип indirect. При непрямом отображении чтение из порта в переменную происходит непосредственно перед очередной обработкой процессов в бесконечном цикле (секция чтения в физические переменные отложенного отображения), а запись значения из переменной в порт соответственно после обработки процессов (секция записи из физических переменных отложенного отображения).

Примеры использования физических переменных прямого и отложенного (непрямого) отображения, а также их правила трансляции в язык Си приведены в таблице 4.

Таблица 4 – трансляция физических переменных в Си

Конструкция	Проекция в язык Си

direct bool d_in_var as input(
  read=PINB,
  config=DDRB,
  bit=PB0
);
	// конфигурация
	SET_BIT_8_INPUT_MODE(DDRB, PB0);

indirect bool ind_in_var as input(
  read=PINB,
  config=DDRB,
  bit=PB0
);
	// объявление
	BOOL ind_in_var;

	// секция чтения в физические переменные
	// отложенного отображения
	ind_in_var = READ_PORT_8_BIT(PINB, PB0);

direct bool d_out_var as output(
  write=PORTB,
  config=DDRB,
  bit=PB0
);
	// конфигурация
	SET_BIT_8_OUTPUT_MODE(DDRB, PB0);

indirect bool ind_out_var as output(
  write=PORTB,
  config=DDRB,
  bit=PB0
);
	// объявление
	BOOL ind_out_var;

	// секция записи из физических переменных // отложенного отображения

	WRITE_PORT_8_BIT(PORTB, ind_out_var , PB0);
if (d_in_var) {...}	if (READ_PORT_8_BIT(PINB, PB0)) {...}
if (! ind_in_var) {...}	if (! ind_in_var) {...}
ind_out_var = ON;	ind_out_var = ON;
d_out_var = OFF;	WRITE_PORT_8_BIT(PORTB, d_out_var, PB0);

2.5. Вставки кода на C

В дополнение к вышеописанным коррекциям в язык Reflex была добавлена возможность вставки кода на языке Си. Такие вставки начинаются символом $ и заканчиваются символом новой строки \n. Этот синтаксис был заимствован из языка Industrial-C. Вставки кода на Си могут использоваться наряду с обычными инструкциями (statement) языка Reflex, и их содержимое транслируется в язык Си без изменений.

2.6. Основные выводы по главе

В данной главе был описан синтаксис и семантика новых конструкций языка Reflex. Расширение языка концептом узлов и легковесных состояний делают язык более гибким и выразительным. Благодаря внедрению в язык возможности работы с регистрами, битами и векторами микроконтроллера, расширяются функциональные возможности языка. Коррекция спецификации отображения переменных на порты ввода-вывода позволила задать как прямое, так и отложенное отображение, что также расширило функциональные возможности языка. Вставка кода на языке Си, заимствованная из Industrial-C, добавила дополнительную гибкость и позволила интегрировать существующий код Си в программы на Reflex.

Глава 3. Имплементация парсера и модуля кодогенерации. Исследование языка на модельных примерах

Для реализации транслятора языка Reflex выбран фреймворк Xtext. Данный фреймворк предоставляет возможности для разработки языков программирования и соответствующих трансляторов, включая возможности для создания синтаксического анализатора (парсера), а также инфраструктуру для интегрированной среды разработки (IDE).

3.1. Архитектура разрабатываемой программной системы

Архитектура разрабатываемой системы включает следующие ключевые компоненты:

- Редактор кода: пользовательский интерфейс для написания и редактирования текстов программ на языке Reflex.
- Парсер. Обрабатывает текст программы, производит синтаксический анализ и создает абстрактное синтаксическое дерево (AST) [11]. При успешном анализе дерево передается семантическому анализатору и модулю кодогенерации. В случае ошибок информация о них возвращается в редактор.
- Семантический анализатор: выполняет проверку типов и областей видимости переменных. В случае обнаружения ошибок информация передается в редактор кода, иначе AST передается модулю кодогенерации.
- Модуль кодогенерации: принимает Синтаксическое дерево, производит обход его узлов и генерирует набор текстовых файлов с определенной структурой.

3.2. Синтаксический анализ программы. Генерация абстрактного синтаксического дерева

3.2.1. Грамматика языка

Грамматика Reflex описывается с помощью языка Xtext Grammar Language [12]. Терминальные правила описываются с помощью выражений, подобных выражениям расширенной формы Бэкуса-Наура (РБНФ) [13].

Для новых конструкций был написан набор соответствующих правил грамматики. В листинге 3 описаны правила для новой конструкции физических переменных языка Reflex.

```
PhysicalVariable:
     mappingType=MappingType? type=Type name=ID "as"
mapping=PortMapping ;

enum MappingType:
     INDIRECT="indirect" | DIRECT="direct";

PortMapping:
     direction=Direction "("
     (("read" "=" readPort=[IdReference] "," "write" "="
writePort=[IdReference])
     | ("read" "=" readPort=[IdReference])
     | ("write" "=" writePort=[IdReference]))
     ("," "config" "=" confPort=[IdReference])?
     ("," "bit" "=" bit=[Bit])? ")";

enum Direction:
     INPUT='input' | OUTPUT='output';
```

Листинг 3 - правила грамматики для объявления физических переменных

Полное описание грамматики языка Reflex представлено в приложении А.

3.2.2. Генерация парсера

Для генерации парсера фреймворк Xtext задействует ANTLR с LL(*) алгоритмом парсинга. Грамматика, определенная пользователем, позволяет Xtext автоматически создавать парсер, который производит синтаксический разбор исходного кода и строит абстрактное синтаксическое дерево.

3.2.3. Представление абстрактного синтаксического дерева

В результате синтаксического анализа строится абстрактное синтаксическое дерево, представленное в Xtext в виде иерархии классов модели EMF (Eclipse Modeling Framework) [14]. Каждый узел дерева представляет собой экземпляр соответствующего класса EMF, описывающего конкретную конструкцию языка (например, выражение, оператор, процесс). EMF предоставляет API для навигации и манипуляции объектами модели. Это позволяет обходить узлы AST, модифицировать их, добавлять или удалять элементы. Модификация AST будет активно применяться при реализации модуля генератора, описанного далее.

3.3. Семантический анализ. Валидация и связывание

Семантический анализ включает в себя проверку типов, областей видимости переменных, корректность использования тех или иных синтаксических конструкций. Реализованный средствами фреймворка Xtext, семантический анализ включает в себя проверку типов, областей видимости переменных, корректность использования тех или иных синтаксических конструкций. Основные аспекты семантического анализа в Xtext – это валидация (validation) и связывание (linking) [15].

Валидация реализована в классе ReflexValidator, который наследуется от AbstractDeclarativeValidator. Для введенных в язык конструкций были написаны методы, осуществляющие необходимые проверки. Все описанные методы помечены специальной аннотацией @Check и в качестве аргумента принимают элемент модели. В зависимости от условий, методы класса ReflexValidator генерируют сообщения об ошибках или предупреждениях при помощи специальных методов error и warning, описанных в классе AbstractDeclarativeValidator. Эти сообщения привязываются к указанным элементам модели.

Пример метода класса ReflexValidator, выполняющего проверку выражения присваивания, представлен в листинге 4. Метод принимает элемент модели, соответствующий выражению присваивания. Сначала из выражения берется элемент, в который производится присваивание. Если этот элемент является физической переменной, отображенной на входной на входной порт, то генерируется либо сообщение об ошибке для переменной прямого отображения, либо предупреждение для переменной непрямого отображения. В случае, если элемент является вектором или битом, будет получено сообщение об ошибке. Сгенерированное сообщение привязано к элементу, в который осуществляется присваивание. Таким образом, строка с ошибкой помечается соответствующим символом в редакторе (рисунок 1).

```
@Check def void checkAssignVariable(AssignmentExpression expr) {
  val assignVar = expr.assignVar
  if (assignVar instanceof PhysicalVariable) {
    if (assignVar.isInput) {
      if (assignVar.isDirect){
        error("An attempt to assign value into variable directly
               mapped on input port",
               ePackage.assignmentExpression_AssignVar)
      }
      else {
        warning("An attempt to assign value into variable mapped on
                 input port",
                 ePackage.assignmentExpression_AssignVar)
      }
    }
  }
  else if (assignVar instanceof Vector||assignVar instanceof Bit){
    error("Can't assign values to vectors or bits",
           ePackage.assignmentExpression_AssignVar)
  }
}
```

Листинг 4 – метод валидатора, выполняющий проверку присваивания

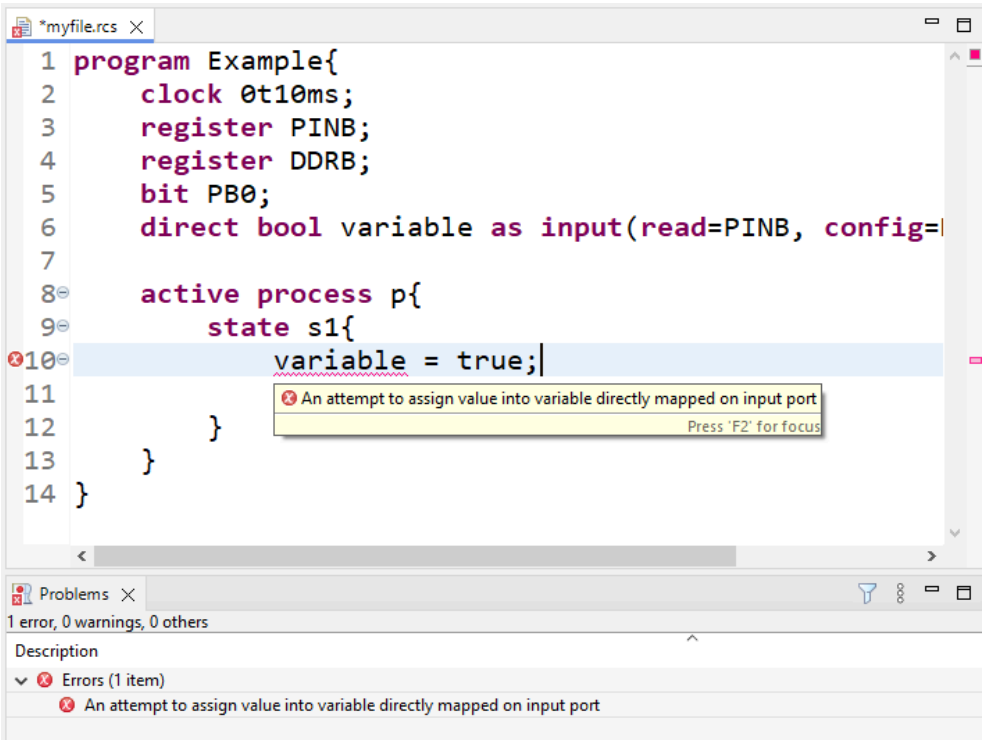

Рисунок 1 – Компоненты системы и их взаимодействие

Связывание (linking, разрешение ссылок) – еще одно средство семантическое анализа в Xtext. Связывание позволяет идентифицировать и находить определения переменных, регистров, битов и других элементов, на которые ссылается код, обеспечивая корректность ссылок и предотвращая ошибки, такие как неразрешенные ссылки или циклические зависимости. Спецификация семантики связываний обеспечивается с помощью scoping API [16].

Основным компонентом Scoping API является интерфейс IScopeProvider, его основной метод getScope предназначен для определения области видимости (scope) для заданного контекста (элемента, содержащего ссылку) и ссылки (reference). Он возвращает объект, реализующий интерфейс IScope, который предоставляет доступные в данной области видимости элементы. Xtext предоставляет стандартную реализацию для IScopeProvider, однако она не подходит для языка Reflex, поэтому был реализован класс ReflexScopeProvider с переопределенным методом getScope.

Рассмотрим следующий пример применения механизма разрешения ссылок. В выражениях, находящихся в контексте состояний процесса, могут использоваться переменные, объявленные в глобальном контексте, контексте процесса или контексте узла, которому принадлежит процесс. Следовательно, область видимости должна включать объявления переменных из всех трех контекстов. В листинге 5 описана реализация ReflexScopeProvider, которая решает эту задачу.

```
class ReflexScopeProvider extends AbstractReflexScopeProvider {
      override IScope getScope(EObject context, EReference ref) {
            if (context instanceof PrimaryExpression &&
                ref == primaryExpression_Reference){
                  State state = context.getContainerOfType(State)
                  Process process = state.getContainerOfType(Process)
                  Program program = state.getContainerOfType(Program)
                  ArrayList<EObject> candidates = newArrayList
                  candidates.addAll(process.variables)
                  candidates.addAll(program.globalVars)
                  candidates.addAll(process.node.globalVars)
                  return Scopes.scopeFor(candidates)
            }
            return super.getScope(context, ref)
      }
}
```

Листинг 5 – разрешение ссылки на переменные в выражениях внутри состояний (псевдокод)

В приложении Б представлен перечень семантических проверок, реализованных для новых конструкций языка Reflex, а также описаны инструменты Xtext, использованные для их внедрения.

3.4. Модуль кодогенерации

Для усовершенствованной грамматики был имплементирован модуль кодогенерации, осуществляющий трансляцию новых конструкций в язык Си. Данный модуль полностью реализован на языке Eclipse/Xtend, гибком и выразительном диалекте Java [17]. Особенно полезная особенность Xtend при реализации кодогенератора – это шаблонные выражения (template expressions),

позволяющие конкатенировать строки в удобном для чтения виде [18]. Пример использования шаблонных выражений языка Xtend представлен в листинге 6.

```
override String generateInput() {
            return '''
            /*==vvvv== Generated Input ==vvvv==*/
          «FOR entry :
portGenerationHelper.getInputIndirectVarsMap(program).entrySet»
            «IF entry.value.size == 1»
                  «identifiersHelper.getMapping(entry.value.get(0))» =
«portGenerationHelper.generateReadFunction(entry.value.get(0))»;
            «ELSE»
                  «identifiersHelper.getRegisterImageId(entry.key)» =
«portGenerationHelper.generateReadFunction(entry.key)»;
                  «FOR v : entry.value»
                  «identifiersHelper.getMapping(v)» =
«portGenerationHelper.generateReadFunction(identifiersHelper.getRegiste
rImageId(entry.key), v.mapping.bit?.name)»;
                  «ENDFOR»
                  «ENDIF»
          «ENDFOR»
            /*==^^^^== End of Input ==^^^^==*/
            '''
```

Листинг 6 - пример кода на языке Xtend

Вся логика по работе с отображением портов для новых физических переменных была вынесена в отдельный класс PortGenerationHelper. Для взаимодействия с регистрами ввода-вывода микроконтроллера используются макросы, определенные в зависимости от платформы в файле platform.h. Пример файла platform.h для работы с микроконтроллерами семейства AVR представлен в листинге 7.

```
#ifndef PLATFORM_H
#define PLATFORM_H
#include <avr/io.h>
#include <avr/interrupt.h>
#define SET_BIT_8_INPUT_MODE(DDRX, PXn) (DDRX &= ~(1 << PXn))
#define SET_PORT_8_INPUT_MODE(DDRX) (DDRX = 0)
#define SET_PORT_8_OUTPUT_MODE(DDRX) (DDRX = 0xFF)
#define SET_BIT_8_OUTPUT_MODE(DDRX, PXn) (DDRX |= (1 << PXn))
#define READ_PORT_8_BIT(PINX, PXn) (((PINX >> PXn) & 1) ? TRUE : FALSE)
#define READ_PORT_8(PINX) (PINX)
#define WRITE_PORT_8_BIT(PORTX, var, PXn) PORTX = ((var) ? (PORTX | (1
<< PXn)) : (PORTX & ~(1 << PXn)))
#define WRITE_PORT_8(PORTX, var) PORTX = (var)
#define WRITE_PORT_8_MASK(port, portImg, mask) port &= ((portImg) |
~(mask)); port |= ((portImg) & (mask))
#endif
```

Листинг 7 – содержимое platform.h для микроконтроллеров ATmega

За генерацию кода программы отвечает класс новый класс R2CProgramGenerator. Класс получает на вход элемент программы (Program), и возвращает текст сгенерированной программы на Си. В свою очередь, в классе R2CFileGenerator были добавлены методы, которые для каждого узла с ненулевым количеством процессов генерируют копию дерева программы, удаляют из этого дерева все элементы, не относящиеся к данному узлу, и вызывают R2CProgramGenerator для получившейся программы.

Для легковесных состояний также модифицируется AST программы. Каждый раз, когда встречается slice в теле состояния, конструкция slice заменяется конструкцией set next state, и все последующие инструкции, описанные для этого состояния, переносятся в новое созданное состояние, новое состояние добавляется в список состояний процесса в соответствующем месте (листинг 8). Для transition и transition on timeout механизм генерации состояний аналогичен.

```
 protected def prepareSlicedStates(Process process) {
   var sliceCounter = 0
   val reflexFactory = ReflexPackage.eINSTANCE.reflexFactory
   var stateIter = process.states.listIterator
   while(stateIter.hasNext){
     val currentState = stateIter.next()
     val statements = currentState.stateFunction.statements
     if (!statements.filter(SliceStatement).isEmpty){
       sliceCounter += 1
       var newState = reflexFactory.createState
       newState.name =  "slice_" + sliceCounter
       var firstSlice = statements.findFirst[e | e instanceof
SliceStatement]
       val firstEntrance = statements.indexOf(firstSlice)
       val copy = statements.subList(firstEntrance,
statements.size).tail.clone()
       statements.remove(firstEntrance)
       newState.stateFunction = reflexFactory.createStatementSequence
       newState.stateFunction.statements.addAll(copy)

       val setStateStat = reflexFactory.createSetStateStat
       setStateStat.next = true
       currentState.stateFunction.statements.add(setStateStat)

       if (currentState.timeoutFunction !== null){
         newState.timeoutFunction = currentState.timeoutFunction
       }
       stateIter.add(newState)
       stateIter.previous
     }
   }
 }
```

Листинг 8 – метод для генерации состояний, создаваемых конструкцией slice

3.4.1. Структура модуля кодогенерации

На рисунке 2 изображена диаграмма основных классов модуля кодогенерации. Диаграмма не включает в себя классы-помощники, такие как PortGenerationHelper.

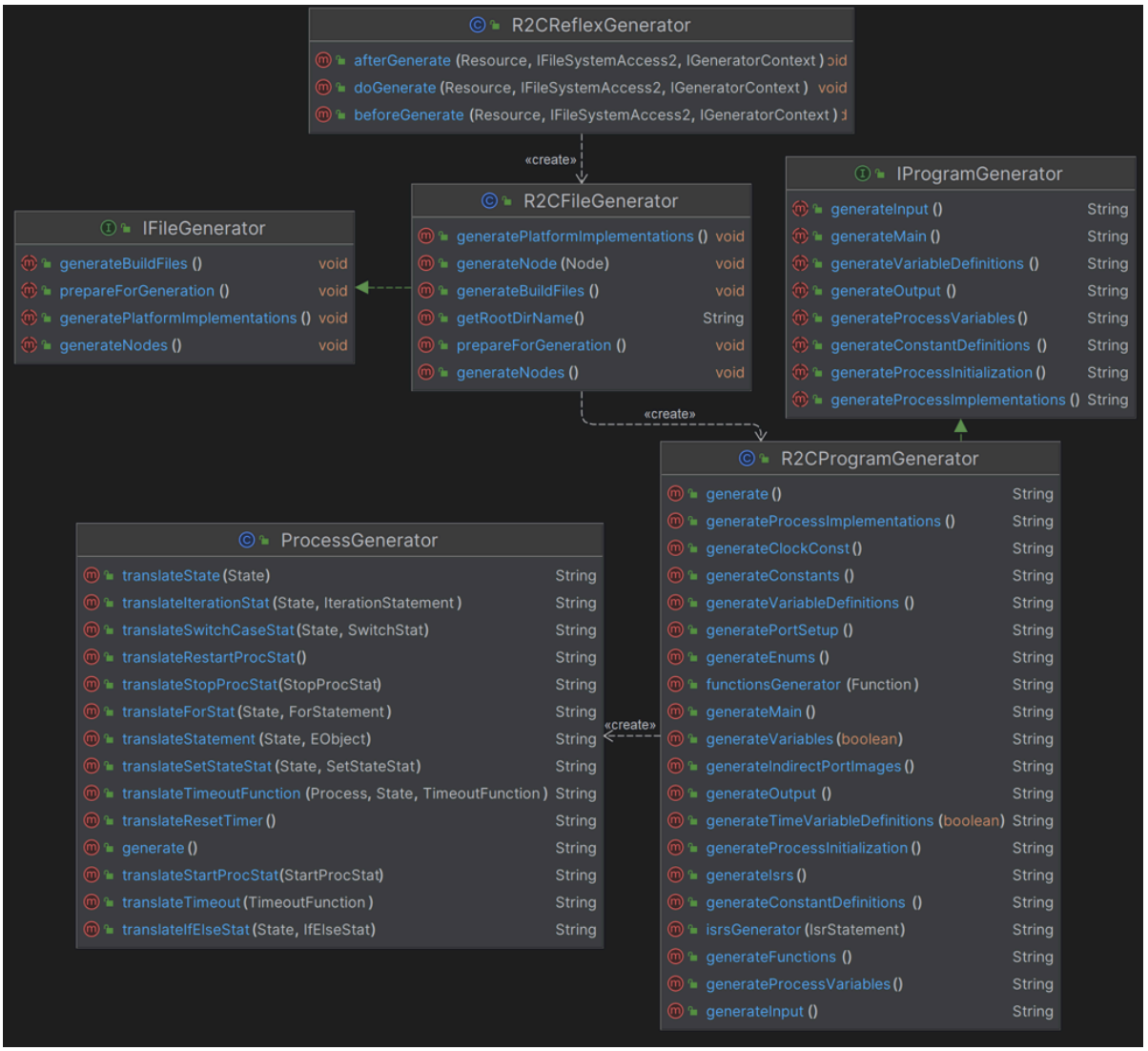

Рисунок 2 – Диаграмма основных классов модуля кодогенерации

3.5. Исследование на модельных примерах

Для контроля корректности трансляции проводилась отладка кодогенератора на модельных примерах с использованием ATmega для среды Arduino. Один из таких примеров – программа «Умная лампа» на языке Reflex (листинг 9), результат трансляции описан в приложении В. Все внесенные корректировки и расширения языка Reflex транслировались корректно. Помимо этого, на модельных примерах были исследованы введенные конструкции языка, описан эффект от использования данных конструкций (таблица 5).

Таблица 5 – полученный эффект от использования введенных в Reflex конструкций

Конструкция в языке Reflex	Эффект
node	возможность описывать распределенную микроконтроллерную встраиваемую систему централизованно
slice;	сокращение трудоемкости – генерируется три/четыре строки, автогенерация идентификатора состояния, возможность введения инициализирующих секций
transition;	сокращение трудоемкости – автогенерация строк (7–8), автогенерация состояний (1–2)
transition on timeout;	сокращение трудоемкости – автогенерация строк (14–15), автогенерация состояний (1–2)
direct/indirect, **as input/output,** **read, write, config**	автогенерация конфигурации, секций считывания/записи переменных, возможность выбора стратегии (синхронная/асинхронная) работа с портами ввода-вывода

ЗАКЛЮЧЕНИЕ

По итогам работы были получены следующие результаты:

1) Синтаксис Reflex расширен новыми конструкциями, делающими язык более функциональным и лаконичным.
2) При помощи фреймворка Xtext реализованы парсер и модуль кодогенерации для усовершенствованной версии языка Reflex.
3) Проведен контроль корректности реализованных средств, исследована эффективность внедренных конструкций.

Таким образом, все поставленные задачи были выполнены, цель работы достигнута. В дальнейшем планируется продолжать поддержку и развитие языка Reflex.

Результаты данной работы докладывались на Международной научной студенческой конференции 2024 в секции «Инструментальные и прикладные программные системы».

Выпускная квалификационная работа выполнена мной самостоятельно 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 недопуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Бурдэ Сергей Петрович _____

ФИО студента *Подпись студента*

« ____ »_____2024 г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Зюбин, В. Е. Язык «Рефлекс» – диалект Си для программируемых логических контроллеров // Шестая международная научно-практическая конференция «Средства системы автоматизации» CSAF. – 6 с.
2. Розов, А. С. Адаптация процесс-ориентированного подхода к разработке встраиваемых микроконтроллерных систем / А.С. Розов, В.Е. Зюбин // Автометрия. – 2019. – Т. 55. – № 2. – С. 114-122.
3. Zyubin, V. E. Hyper-automaton: A Model of Control Algorithms / V.E. Zyubin // Материалы международной научной конференции IEEE International Siberian Conference on Control and Communications (SIBC0N-2007). – 2007. – С. 51-57.
4. Зюбин, В. Е. Процесс-ориентированное программирование: Учеб. пособие/ В. Е.Зюбин – Новосиб. гос. ун-т. – 2011. – 194 с.
5. Розов А. С. и др. Практическая апробация языка IndustrialC на примере автоматизации установки термовакуумного напыления //Вестник Новосибирского государственного университета. Серия: Информационные технологии. – 2017. – Т. 15. – №. 3. – С. 90-99
6. Краснов Д. В. и др. Практическая апробация процесс-ориентированной технологии программирования на открытых микроконтроллерных платформах // Вестник ВСГУТУ. 2017. Т. 66, вып. 3. – С. 85–92.
7. Лях Т. В., Зюбин В. Е., Сизов. М. М. Опыт применения языка Reflex при автоматизации Большого солнечного вакуумного телескопа // Промышленные АСУ и контроллеры. 2016. № 7. – С. 37-43.
8. IEC 61131-3 Programmable controllers. Part 3: Programming languages // International Electrotechnical Commission. 2003. – 332 p.
9. Levine, John. Flex & Bison: Text Processing Tools. // O'Reilly Media, Inc., 2009. – 544 p.
10. Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." In Proceedings of the ACM international

conference companion on Object oriented programming systems languages and applications companion. ACM, 2010. – pp. 307-309

11. Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. // Pearson, 2007. – 1009 p.
12. Eclipse Foundation. Xtext Documentation: Grammar Language [Электронный ресурс]. URL: https://eclipse.dev/Xtext/documentation/301_grammarlanguage.html (дата обращения 16.05.2024).
13. Perugini, Saverio. Programming Languages: Concepts and Implementation. // Jones & Bartlett Learning, 2021. – 658 p.
14. Eclipse Foundation. Xtext Documentation: EMF Integration [Электронный ресурс]. URL: https://eclipse.dev/Xtext/documentation/308_emf_integration.html (дата обращения 16.05.2024).
15. Eclipse Foundation. Xtext Documentation: Runtime Concepts. Validation [Электронный ресурс]. URL: https://eclipse.dev/Xtext/documentation/303_runtime_concepts.html#validation (дата обращения 16.05.2024).
16. Eclipse Foundation. Xtext Documentation: Runtime Concepts. Scoping [Электронный ресурс]. URL: https://eclipse.dev/Xtext/documentation/303_runtime_concepts.html#scoping (дата обращения 16.05.2024).
17. Bettini L. Implementing domain-specific languages with Xtext and Xtend. – Packt Publishing Ltd, 2016. – 420 p.
18. Eclipse Foundation. Xtext Documentation: Xtend Expressions. Templates [Электронный ресурс]. URL: https://eclipse.dev/Xtext/xtend/documentation/203_xtend_expressions.html#templates (дата обращения 16.05.2024).

ПРИЛОЖЕНИЕ А

Описание грамматики Reflex

```
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
     (imports+=Import |
     registers+=Register |
     bits+=Bit |
     vectors+=Vector |
     consts+=Const |
     enums+=Enum |
     functions+=Function |
     globalVars+=GlobalVariable |
     processes+=Process)*
     "}";
ClockDefinition:
     "clock" (intValue=INTEGER | timeValue=TIME) ";";
Import:
     "import" name=ID "{"
     (registers+=Register |
     bits+=Bit |
     vectors+=Vector)* "}";
Vector:
     "vector" size=Type? name=ID ";";
Register:
     "register" size=Type? name=ID ";";
Bit:
```

```
     "bit" name=ID ";";
RegisterReference:
     Const | Register;
Process:
     ("[" annotations+=Annotation "]")*
     (active?="active")? "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:
     mappingType=MappingType? type=Type name=ID "as"
mapping=PortMapping ;

enum MappingType:
     INDIRECT="indirect" | DIRECT="direct";

PortMapping:
     direction=Direction "("
     (("read" "=" readPort=[RegisterReference] "," "write" "="
writePort=[RegisterReference])
     | ("read" "=" readPort=[RegisterReference])
     | ("write" "=" writePort=[RegisterReference]))
     ("," "config" "=" confPort=[RegisterReference])?
     ("," "bit" "=" bit=[Bit])? ")";

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

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)* ")" ";";

Const:
```

```
     "const" type=Type name=ID "=" value=Expression ";";

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

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

     // Statements + C code insertion
Statement:
     {Statement} ";" | CompoundStatement |
     StartProcStat | StopProcStat | ErrorStat | RestartStat | 
ResetStat
     | SetStateStat | IfElseStat | SwitchStat | Expression ";" | 
C_code;

StatementSequence:
     {StatementSequence} statements+=Statement*;

// C code insertion
C_code:
     {C_code} c_code += C_PHRASE;

terminal C_PHRASE:
    '$' (!'\n')* ('\n');
// +  '$' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'$'|"'"|'\\' */ | 
!('\\'|'$') )* '$';
// -   '$' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'$'|"'"|'\\' */ | 
!('\\'|'$') )* ('\r'? '\n');
//   '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | 
!('\\'|'"') )* '"'

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:
     INT8="int8" | VOID_C_TYPE="void" | FLOAT="float" |
DOUBLE="double" | 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)?;

terminal FLOAT:
     DEC_FLOAT | HEX_FLOAT;

terminal fragment DEC_FLOAT:
     SIGN? 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 Б

Перечень семантических проверок языка Reflex

Таблица Б.1 – перечень семантических проверок языка Reflex

Проверка	Реакция на нарушение правила	Средство Xtext, используемый для реализации проверки
Проверка размера регистров	Ошибка	validation
Проверка типа переменной, отображенной на порт (целиком или частично)	Ошибка	validation
Проверка предоставления регистра чтения для входного порта	Ошибка	validation
Проверка предоставления регистра записи для выходного порта	Ошибка	validation
Проверка отображения переменной на одни и те же регистры	Предупреждение	validation
Проверка записи в переменную прямого отображения отображенную на входной порт	Ошибка	validation
Проверка записи в переменную непрямого отображения отображенную на входной порт	Предупреждение	validation
Проверка записи в регистр или бит	Ошибка	validation
Контроль доступности регистров, битов, векторов определенных в блоках import	Ошибка	linking
Проверка использования slice и transition только среди выражений верхнего уровня состояний	Предупреждение	validation
Проверка использования	Предупреждение	validation

инструкций перехода состояний до slice или transition		
Проверка двух slice подряд	Предупреждение	validation
Проверка имени узла	Предупреждение	validation
Контроль доступности переменных, констант, перечислений узла	Ошибка	linking

ПРИЛОЖЕНИЕ В

Листинг программы «Умная лампа» на Си

```
#include "platform.h"
INT32_U _r_cur_time;
INT32_U _r_next_act_time;
#define ON TRUE
#define OFF FALSE
#define TIMEOUT (INT32_U) 30000UL
#define pinb 0x25
#define _r_CLOCK (INT32_U) 10UL
/* ===== process state constants ====== */
#define _START 0
#define _CONFIRMED 253
#define _ERROR 254
#define _STOP 255
/* ===== process state vars ====== */
uint8_t _p_Light_switching_state;
/* ===== process timer vars ====== */
uint32_t _p_Light_switching_time;
/* ===== process states enumerators ====== */
enum _p_Light_switching_states {
 _p_Light_switching_s_Wait = _START,
 _p_Light_switching_s_Work
};
void setup() {   /* Init */
 /* Ports configuration */
 SET_BIT_8_INPUT_MODE(DDRB, PB0);
 SET_BIT_8_OUTPUT_MODE(DDRB, PB1);
 _r_cur_time = millis();
 _r_next_act_time = _r_cur_time;
 // #include "port_init.h"
 /*========= PROCESS INIT: ========*/
```

```
 _p_Light_switching_state = _START;
 /*========= END OF PROCESSES INIT========*/
}
void loop() {   /* Control algorithm */
 _r_cur_time = get_time();
 if (_r_cur_time - _r_next_act_time >= 0) {
   // Find next activation time
   _r_next_act_time += _r_CLOCK;
   if (_r_next_act_time - _r_cur_time > _r_CLOCK) {
     _r_next_act_time = _r_cur_time + _r_CLOCK;
   }
   /*========= PROCESS IMAGES: ========*/
   //== Process "Light_switching":
   switch (_p_Light_switching_state) {
     case _p_Light_switching_s_Wait: { /* State: Wait */
       if (READ_PORT_8_BIT((INT8_U*)pinb, PB0))
         {
         WRITE_PORT_8_BIT(PORTB, ON, PB1);
         _p_Light_switching_state = _p_Light_switching_s_Work;
         _p_Light_switching_time = _r_cur_time;
         }
       break;
     }
     case _p_Light_switching_s_Work: { /* State: Work */
       if (READ_PORT_8_BIT((INT8_U*)pinb, PB0))
         _p_Light_switching_time = _r_cur_time;
       if ((_r_cur_time - _p_Light_switching_time) > TIMEOUT) // timeout
         {
         WRITE_PORT_8_BIT(PORTB, READ_PORT_8_BIT((INT8_U)pinb, PB0),
PB1);
         _p_Light_switching_state = _p_Light_switching_s_Wait;
         }
       break;
```

```
      }
    }
    /*========== END OF PROCESSES ========*/
  }
}
```

ПРИЛОЖЕНИЕ Г

Eclipse IDE языка Reflex

Руководство оператора

Листов 6

Новосибирск 2024

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации программной системы, представленной в виде Eclipse IDE процесс-ориентированного языка Reflex.

В разделе «Назначение программы» содержатся сведения о предназначении программы и перечислены ее функции.

Раздел «Условия выполнения программы» включает требования, необходимые для её корректного выполнения.

В разделе «Выполнение программы» описана последовательность действий оператора, обеспечивающая загрузку, запуск, выполнение и завершение работы программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

1. Назначение программы

Программная система предназначена для использования процесс-ориентированного языка программирования Reflex с помощью Eclipse IDE.

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Отображение файловой системы;
- Создание / удаление файлов;
- Редактирование файлов;
- Подсветка синтаксиса;
- Выделение ошибок и предупреждений;
- Автодополнение;
- Навигация по коду;
- Поиск по коду;
- Синтаксическая и семантическая проверка;
- Список ошибок и предупреждений, с возможностью перехода в проблемный участок кода;
- Отображение структуры открытого файла;
- Генерация Си кода ;

2. Условия выполнения программы

2.1. Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере. Минимальный перечень технических средств, обеспечивающих работу программы:

1) Оперативная память объемом не менее 8 Гб;
2) Процессор, совместимый с архитектурой x86, и тактовой частотой 2 ГГц или выше;
3) Жёсткий диск объёмом не менее 250 Гб;
4) Лицензионная операционная система, unix-подобная или windows;

2.2. Требование к персоналу

Конечный пользователь программы (оператор) должен иметь практические навыки работы со средой разработки Eclipse, уметь создавать и запускать Xtext проекты.

3. Выполнение программы

3.1. Загрузка и запуск программы

Запуск Eclipse IDE. Открытие проекта через File, Open Projects from File System. Выбор директории проекта Reflex и нажатие Finish. В Project Explorer открытие файла с расширением .reflex. Проверка синтаксиса и сохранение файла для генерации кода (Ctrl+S).

3.2. Выполнение программы

В Project Explorer проверка сгенерированных файлов (обычно в src-gen). В меню Run, Run Configurations создание новой конфигурации запуска. Выбор проекта и главного класса для выполнения. При необходимости указание аргументов командной строки на вкладке Arguments. Нажатие Apply и Run. Наблюдение за выводом программы в Console view.

3.3. Завершение работы программы

Остановка выполнения программы через кнопку Terminate в Console view или Run, Terminate в меню. Проверка завершения выполнения программы и выводов на ошибки. Закрытие проекта через Project Explorer, если работа завершена.