Информатика. Экономика. Управление//
Informatics. Economics. Management

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

УДК: 004.4
DOI: https://doi.org/10.47813/2782-5280-2022-1-1-0201-0216

Инструментальная поддержка интерпретации и динамической компиляции языков программирования систем промышленной автоматизации

А. В. Дроздов

Сибирский Федеральный Университет, кафедра информатики Института космических и информационных технологий, Красноярск, Российская Федерация

Аннотация. Работа посвящена исследованию различных реализаций сред исполнения для языков промышленной автоматизации стандарта МЭК 61131-3 и проектированию среды исполнения, отличающейся от аналогов подходом к трансляции и выбором целевой платформы. Рассмотрена модель реализации, включающая в себя синтаксический разбор программного кода на языке ST стандарта МЭК 61131-3 посредством алгоритма LALR, последующая его интерпретация на виртуальной машине Java и динамическая компиляция в Java байткод. Рассмотрены и спроектированы части виртуальной машины, специфичные для языков промышленной автоматизации – планировщик задач и модуль управления конфигурацией подключаемых устройств. Разработана модель взаимодействия частей программы с применением архитектуры на основе плагинов.

Ключевые слова: виртуальная машина, среда исполнения, промышленная автоматизация, МЭК 61131-3, интерпретация, компиляция, динамическая компиляция, байткод, автоматизированное управление.

Для цитирования: Дроздов, А. В. (2022). Инструментальная поддержка интерпретации и динамической компиляции языков программирования систем промышленной автоматизации. Информатика. Экономика. Управление - Informatics. Economics. Management, 1(1), 0201–0216. https://doi.org/10.47813/2782-5280-2022-1-1-0201-0216

Instrumental support for interpretation and dynamic compilation of programming languages for industrial automation systems

A. V. Drozdov

Siberian Federal University, Department of Informatics, Institute of Space and Information Technologies, Krasnoyarsk, Russian Federation

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Abstract. The work is devoted to the study of various implementations of runtime environments for industrial automation languages of the IEC 61131-3 standard and the design of a runtime environment that differs from analogues in its approach to translation and the choice of the target platform. The implementation model is considered, which includes parsing the program code in the ST language of the IEC 61131-3 standard using the LALR algorithm, its subsequent interpretation on the Java virtual machine and dynamic compilation into Java bytecode. Parts of the virtual machine specific to industrial automation languages are considered and designed - the task scheduler and the module for managing the configuration of connected devices. A model for the interaction of parts of the program using a plugin-based architecture has been developed.

Keywords: virtual machine, runtime environment, industrial automation, IEC 61131-3, interpretation, compilation, dynamic compilation, bytecode, automated control.

For citation: Drozdov A. V. (2022). Instrumental support for interpretation and dynamic compilation of programming languages for industrial automation systems. Informatics. Economics. Management, 1(1), 0201–0216. https://doi.org/10.47813/2782-5280-2022-1-1-0201-0216

ВВЕДЕНИЕ

Программируемые логические контроллеры (ПЛК) используются практически во всех сферах человеческой деятельности для автоматизации технологических процессов, в системах противоаварийной защиты и сигнализации, в станках с числовым программным управлением, для управления дорожным движением, в системах жизнеобеспечения зданий, для сбора и архивирования данных, в системах охраны, в медицинском оборудовании, в системах связи, при постановке физического эксперимента, для автоматизации испытаний продукции и т. д.

В настоящие время для разработки программ для ПЛК используются в основном языки стандарта МЭК 61131-3 (далее МЭК-языки) [1]. В состав стандарта МЭК 61131-3 входит 5 независимых языков:

- Instruction List (Список Инструкций) – Текстовый язык. Аппаратно-независимый низкоуровневый ассемблероподобный язык;
- Ladder Diagram (Релейно-Контактные Схемы) – Графический язык. Представляет собой программную реализацию электрических схем на базе электромагнитных реле;
- Function Block Diagram (Функциональные Блоковые Диаграммы) – Графический язык. Программа образуется из списка цепей, выполняемых последовательно сверху вниз;

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

- Sequential Function Chart (Последовательные Функциональные Диаграммы) – Графический язык. Создан на базе математического аппарата сетей Петри. Описывает последовательность состояний и условий переходов;
- Structured Text (Структурированный текст) – Текстовый язык. По структуре и синтаксису ближе всего к языку программирования Паскаль.

В большинстве комплексов программирования для трансляции МЭК-языков используется принцип статической компиляции. В некоторых случаях это неприемлемо, т.к. статический компилятор генерирует машинный код только для конкретной архитектуры. Такой машинный код является архитектурно-зависимым и поэтому его использование на других платформах невозможно. Это значит, что для каждой используемой архитектуры необходимо разрабатывать отдельный компилятор, что является достаточно дорогим решением, если необходимо поддерживать несколько архитектур.

JIT-компиляция построена на использовании принципов динамической компиляции и так называемого промежуточного представления (байт-кода). JIT-система транслирует исходный код программы в байт-код, который затем используется для генерации машинного кода во время исполнения исходной программы.

Байт-код является архитектурно-независимым, что обеспечивает его переносимость на платформы с различными архитектурами, где есть соответствующий JIT-компилятор. Разработка JIT-компилятора для байт-кода требует меньших затрат, по сравнению с реализацией статического компилятора для исходного программного кода, поскольку байт-код имеет низкоуровневую структуру.

Более того, использование байт-кода значительно упрощает процесс построения перенастраиваемого JIT-компилятора, т.е. такого компилятора, который позволяет генерировать код для нескольких целевых архитектур, а процесс генерации машинного кода из байт-кода гораздо быстрее, чем из исходного.

В данной статье мы рассмотрим существующие модели виртуальных машин для языков программирования стандарта МЭК 61131-3 и предложим архитектуру виртуальной машины, сочетающей принципы кроссплатформенной интерпретации и динамической компиляции программного кода в машинно-независимое промежуточное представление.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

ВИРТУАЛЬНЫЕ МАШИНЫ IEC 61131-3

На данный момент, стандарт IEC 6113-1 имеет несколько доступных кроссплатформенных реализаций. Рассмотрим некоторые из них:

1. CODESYS Control SoftPLC преобразует любое встроенное устройство или устройство на базе персональных компьютеров (далее ПК) в промышленный контроллер, совместимый с МЭК 61131-3 [2]. Эта система среды выполнения включает в себя важные дополнительные функции, чтобы контроллер мог взаимодействовать с другими компонентами в среде автоматизации. Поддерживает большинство современных архитектур процессоров, включая x64, x86 и ARM.

Реализации CODESYS Runtime для различных платформ являются платными. Лицензионные копии могут быть приобретены на сайте разработчика.

2. The PLCNext Runtime позволяет запускать программы, написанные на языках программирования стандарта МЭК 61131-3, на платформе Microsoft ECLR (Embedded Common Language Runtime) [3]. PLCNext Runtime доступен для скачивания на ПК с операционной системой Linux, Windows, а также на ПЛК PLCNext Control производства Pheonix Contact.

Разрабатываемый продукт представляет собой набор программного обеспечения для осуществления интерпретации и динамической компиляции программ, разработанных с использованием языка программирования ST стандарта МЭК 61131-3 для систем промышленной автоматизации.

В качестве целевой платформы для реализации будет использоваться виртуальная машина Java (далее JVM). Независимость виртуальной машины Java от конкретной аппаратной платформы может значительно увеличить количество новых устройств, способных исполнять написанные ранее сценарии на языках стандарта МЭК 61131-3.

Предлагаемое решение будет отличаться от аналогов использованием различной целевой платформы, альтернативным подходом к трансляции, а также ценовой политикой в отношении распространения программного продукта (таблица 1).

Таблица 1. Сравнительная характеристика аналогов и планируемого программного изделия.

Table 1. Comparative characteristics of analogues and the planned software product.

Показатели	CODESYS Runtime	PLCNext Runtime	Своя разработка

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Совместимость с Windows	+	+	+
Совместимость с Java	-	-	+
Целевая платформа	Custom	ECLR	JVM
Метод трансляции	AOT to bytecode	AOT to MSIL	интерпретация + JIT to bytecode
Бесплатное использование	-	+	+
Поддерживаемые языки	ST/LD/FD/SFC/IL	ST/LD/FD/SFC/IL	ST

ПРОЕКТИРОВАНИЕ ВИРТУАЛЬНОЙ МАШИНЫ

Разрабатываемая виртуальная машина состоит из нескольких архитектурных модулей:

- Модуль пользовательского интерфейса (UI);
- Лексико-синтаксический анализатор языка ST (Parser);
- Интерпретатор абстрактного синтаксического дерева программы (Interpreter);
- JIT-компилятор (JIT);
- Модуль взаимодействия с окружением.

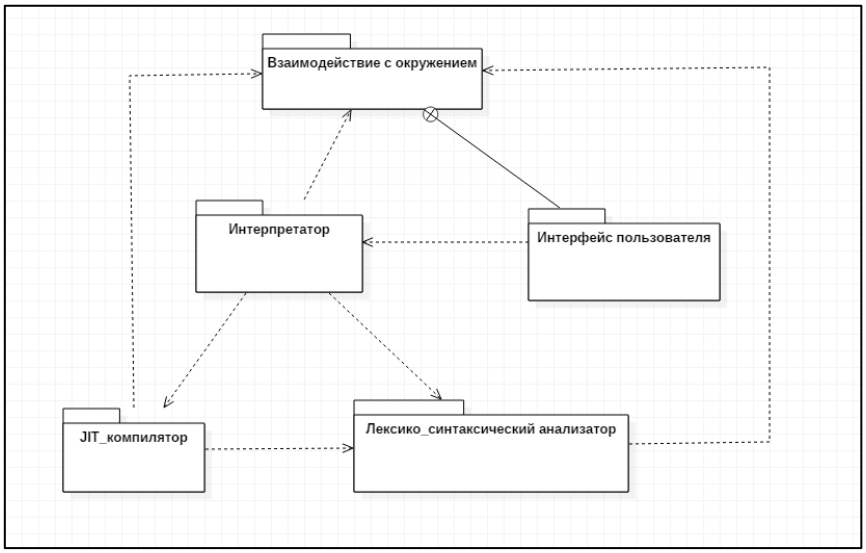

Рисунок 1. Диаграмма пакетов.

Figure 1. Packet diagram.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Зависимости между указанными выше модулями представлены на диаграмме пакетов на рисунке 1.

Основной сценарий работы приложения изображен на диаграмме последовательностей (рисунок 2).

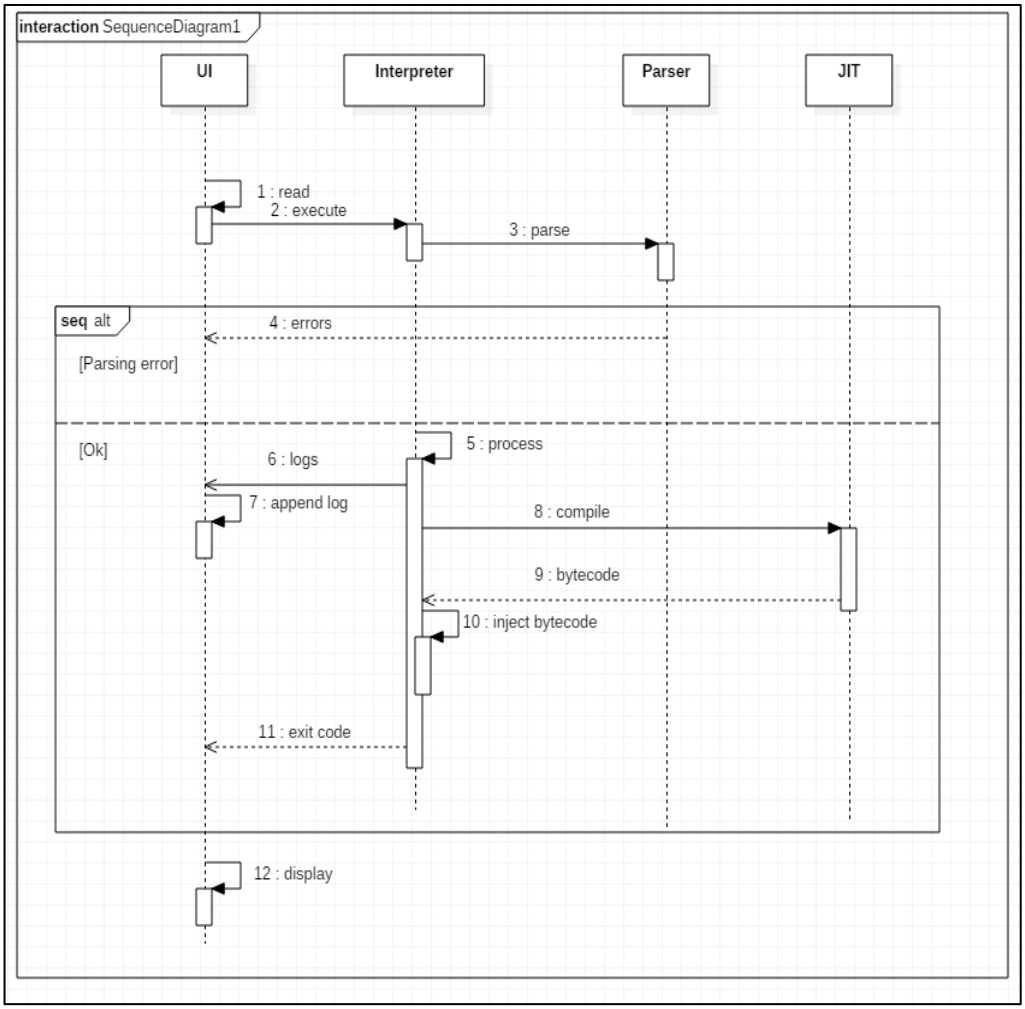

Рисунок 2. Диаграмма последовательностей.

Figure 2. Sequence diagram.

Далее каждый из модулей будет рассматриваться отдельно.

ВЫБОР СРЕДСТВА ПОСТРОЕНИЯ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

В полной версии стандарта МЭК 61131-3, опубликованной на сайте, приводится описание грамматики языков, вошедших в стандарт [4]. Грамматика выбранного для реализации языка ST является LALR-совместимой и содержит всего несколько сотен

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

правил [5]. Для перевода формальных грамматических правил в текст программы был выбран подход с использованием парсер-генераторов. Ниже приведен анализ используемых инструментов для языка программирования Java:

1. JavaCC;
2. ANTLR;
3. SableCC;
4. GNU Bison + flex;

JavaCC – по заявлению разработчиков, наиболее популярный инструмент для построения синтаксических анализаторов на Java. Грамматики выстраиваются по методу нисходящего синтаксического анализа. Прост в использовании с LL(k)-грамматиками. Не имеет встроенного механизма для работы с LALR-грамматиками, так же как и средств для распознавания “shift-reduce”, “reduce-reduce” конфликтов [6].

ANTLR – генератор нисходящих (LL(*)) анализаторов для формальных языков. Имеет широкую сферу использования в продуктах Java-разработки и развитое сообщество. (Hibernate HQL, Cassandra, Groovy). Имеет собственную среду разработки, систему плагинов для популярных сред разработки (IntelliJ IDEA, Eclipse). Обладает вышеперечисленными недостатками генераторов LLпарсеров [7].

SableCC – объектно-ориентированный фреймворк для построения компиляторов. Работает с LALR-грамматиками, описанными в расширенной форме Бэкуса-Наура (РБНФ) [8]. Нотация грамматических правил дополнена метасимволами (*, +, ?) для операторов, заимствованных из регулярных языков, с соответствующей им семантикой. Классы абстрактного синтаксического дерева (далее – AST), также как и примитивы реализации ООП-шаблона «Посетитель» могут генерироваться автоматически [9].

GNU Bison – генератор синтаксических анализаторов для LALRграмматик. Используется в комплексе с лексическим анализатором flex. Поддерживает ряд языков программирования, включая C, C++ и Java [10].

При выборе наиболее подходящего инструмента учитывался собственный опыт работы с Yacc для LALR-грамматик с использованием языка OCaml, количество необходимого подготовительного кода, простота внесения изменений в процесс синтаксического анализа, качество документации и рекомендации специалистов.

По первому критерию, предпочтение отдавалось генераторам синтаксических анализаторов на основе РБНФ-текстов с функцией автоматического распознавания LALR-грамматик (SableCC и Bison). Оба продукта являются открытыми (Bison

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

распространяется по лицензии GPL, SableCC - LGPL), документация доступна публично. Количество подготовительного кода у SableCC меньше, а AST генерируется автоматически. На основании проведенного сравнительного анализа вышеперечисленных инструментов выбор пал в пользу SableCC.

МОДУЛЬ ЛЕКСИКО-СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Модуль лексико-синтаксического анализа представлен классом Parser, реализующим метод parse. Метод parse выполняет синтаксической разбор текста и выполняет построение абстрактного синтаксического дерева программы.

Описание абстрактного синтаксического дерева программы представлено в виде иерархии классов на основе абстрактного базового класса Node.

Минимальным примитивом синтаксического дерева является терминальный символ (токен, содержащий значение). Для каждого известного терминала был выделен отдельный класс, с общим предком Token, унаследованным от класса Node.

Последовательности символов (терминалов и нетерминалов) объединены в альтернативы (синтаксические конструкции), а альтернативы – в продукции (синтаксические блоки). Каждая продукция имеет свой абстрактный класс, описывающий структуру своего поддерева. Альтернативы, входящие в продукцию, описывают собственные классы, которые унаследованы от класса продукции. Упрощенный пример такой иерархии представлен на UML диаграмме классов (рисунок 3). Токены помечены префиксом (T-), альтернативы – (A-), продукции – (P-).

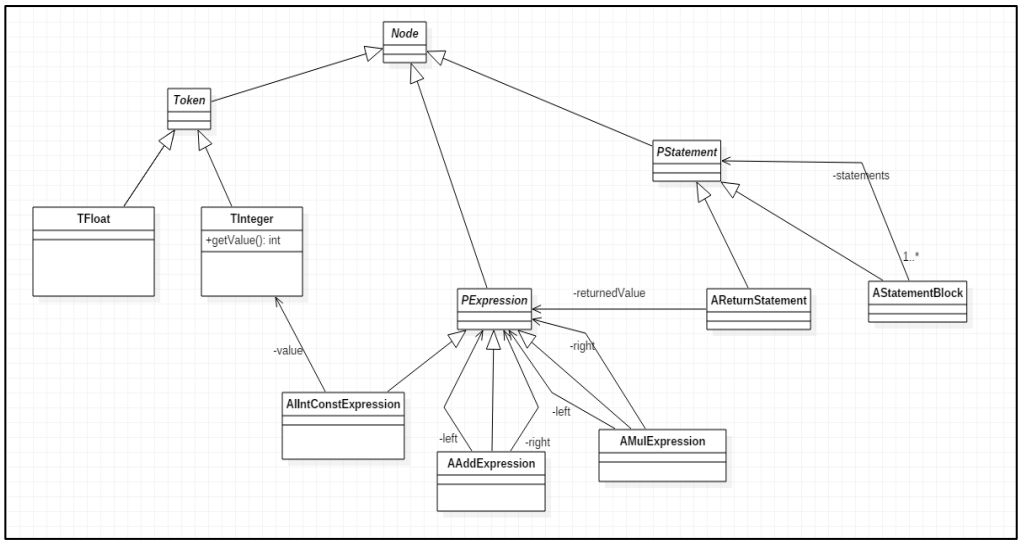

Рисунок 3. Вариант диаграммы классов синтаксического анализатора.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Figure 3. A variant of the parser class diagram.

Построение абстрактного синтаксического дерева начинается с определения правил распознавания терминальных и нетерминальных символов (токенов) и построения лексического анализатора в потоке буквенных символов текста. Здесь используется язык регулярных выражений и таблица переходов между состояниями.

Аналогичным образом определяется синтаксический анализатор. В качестве регулярного языка выступает нотация Бэкуса-Наура (РБНФ) для построения грамматики в терминах продукция и альтернатив. РБНФ оперирует потоком лексем, определенных в ходе лексического анализа [8].

МОДУЛЬ ИНТЕРПРЕТАЦИИ

Модуль интерпретации отвечает за исполнение абстрактного синтаксического дерева программы и осуществляет контроль над временем жизни приложения. Цикл работы основных функций модуля, а также список составных частей представлен на рисунке 4:

- интерпретатор синтаксического дерева;
- планировщик задач;
- конфигуратор устройств;
- подключаемый модуль драйверов устройств.

Интерпретатор синтаксического дерева представлен классом Interpreter с методами execute и process. Метод execute выполняет чтение символов из текстового потока, делегирует синтаксическому анализатору задачу по трансляции текста в абстрактное синтаксическое дерево, управляет добавлением записей в журнал об ошибках, но главным образом – выполняет интерпретацию синтаксического дерева посредством вызова метода process.

Метод process выполняет анализ синтаксического дерева через полную его трассировку. Трассировка реализуется с применением шаблона проектирования «Посетитель» (рисунок 5) [11].

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

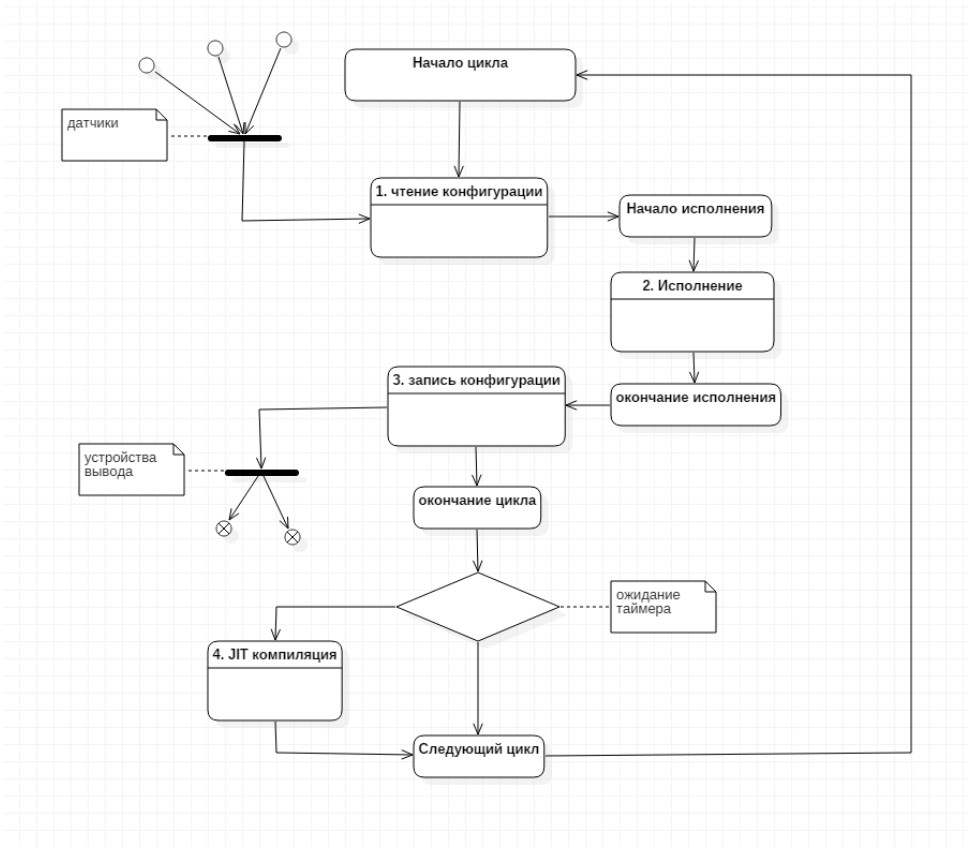

Рисунок 4. Диаграмма жизненного цикла исполняемой программы.

Figure 4. Diagram of the life cycle of an executable program.

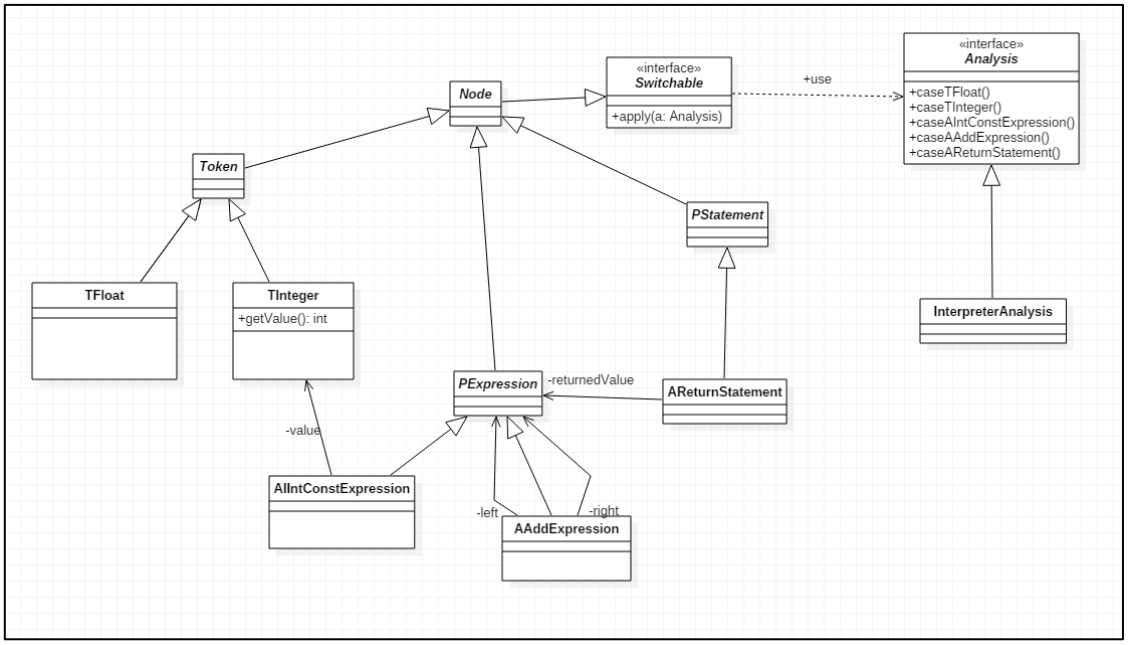

Рисунок 5. Диаграмма классов для модуля интерпретации.

Figure 5. Class diagram for the interpretation module.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Наименования элементов шаблона немного изменены без изменения сути. Каждый узел абстрактного синтаксического дерева реализует интерфейс Switchable с методом apply (в оригинальном изложении шаблона «Посетитель» интерфейс имеет другое название – «Element» с методом accept [11]). Реализация метода сводится к простому вызову соответствующего метода у объекта, переданного в качестве аргумента и реализующего интерфейс «Analysis» («Visitor»). Именно здесь и будет находиться логика интерпретации (исполнения) программного кода, ассоциированного с данным узлом абстрактного синтаксического дерева.

Важным аспектом интерпретации программного кода является контроль над областями видимости и временем жизни компонентов исполняемой программы – переменных и функций. Для обеспечения надежности исполнения, в классе «InterpreterAnalysis» будет использована хэш-таблица с именами и сигнатурами функций и переменных. Своевременное выполнение операций поиска, добавления и удаления привязок к объектам целевой программы позволит решить описанную выше проблему.

Управление памятью времени исполнения осуществляется за счет встроенного в JVM автоматического сборщика мусора [12].

ПЛАНИРОВЩИК ЗАДАЧ

Стандарт МЭК 61131-3 поддерживает распределение вычислительных блоков программы (POU) между различными потоками исполнителя при помощи встроенного механизма задач. Задачи описываются инструкциями на языке ST в блоке конфигурации. Конфигурация задает цикл срабатывания каждой из задач, а также набор переменных окружения, описывающих взаимодействие программы на языке ST с оконечными устройствами автоматизации как в начале выполнения цикла (чтение конфигурации), так и в конце (запись конфигурации).

В описываемой реализации задачи выполняются параллельно в изолированных участках памяти, средствами встроенного в Java механизма Executors [13]. В случае использования глобальных переменных, захваченных при помощи модификатора VAR_EXTERNAL, на поток, обслуживающий задачу, во избежание состояния гонки ставится блокировка до окончания времени выполнения задачи.

Перед началом выполнения задачи происходит чтение данных из подключаемых устройств с занесением в локальные переменные при помощи модуля конфигурации устройств. По окончании цикла, измененные переменные записываются в выходной поток, и обрабатываются тем же модулем.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Допустим, что исполнение программного кода задачи занимает у интерпретатора, с учетом прерываний на конкурентное вычисление, некоторое время T. Чтение и запись занимают дополнительное время R и W.

В идеале, время T + R + W не должно превышать время, отведенное разработчиком на выполнение всего цикла программы, установленного в разделе конфигурации задачи. В противном случае, переход к следующему циклу будет выполнен позже установленного срока.

Если же время после выполнения цикла еще остается, то освободившиеся ресурсы интерпретатора могут быть потрачены на запущенный в фоне JIT-компилятор. Такое распределение ресурсов может быть достигнуто путем выбора настройки планировщика Java Executors.

МОДУЛЬ КОНФИГУРАЦИИ УСТРОЙСТВ

Модуль конфигурации устройств представляет собой набор классов, организующих подключение к конечным устройствам из управляемого кода интерпретатора. Модуль отвечает за чтение и установку конфигурации в начале и конце исполнения цикла задач.

Каждое из устройств, подключенных к системе, должно самостоятельно определять как именно изменение того или иного параметра влияет на состояние его внешних интерфейсов. Описание логики взаимодействия каждого из таких устройств с внешней средой выходит за рамки возможностей языка программирования ST.

Описываемая виртуальная машина предлагает решение проблемы взаимодействия с внешней средой через систему драйверов, т.е. объектов класса, реализующих интерфейс Connector. Исходный код системы драйверов представлен в виде внешней, динамически подключаемой библиотеки (плагина), написанной на языке Java или любом другом совместимом с ней языке. Ответственность за реализацию драйверов, не включенных в поставку виртуальной машины, лежит на разработчике, занимающимся разработкой сценариев промышленной автоматизации. Реализация плагинов теоретически может быть ограничена только степенью поддержки используемых для нее сторонних библиотек со стороны компьютера или ПЛК, на которую установлена данная версия виртуальной машины Java. Например, не исключается возможность использования подсистемы JNI для вызова нативных интерфейсов [14].

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

Плагин должен содержать список классов устройств, реализующих интерфейс Connector (рисунок 6), а также класс Connectors, реализующий статический метод loadConnectors(). Метод loadConnectors() возвращает список устройств, зарегистрированных в системе.

Каждое устройство описывается следующим набором методов:

- getId() – возвращает уникальный идентификатор устройства;
- connect() – устанавливает подключение к устройству;
- close() – закрывает подключения к устройству;
- getConfig() – получает значения для набора переменных конфигурации;
- setConfig() – устанавливает значения конфигурации для устройств.

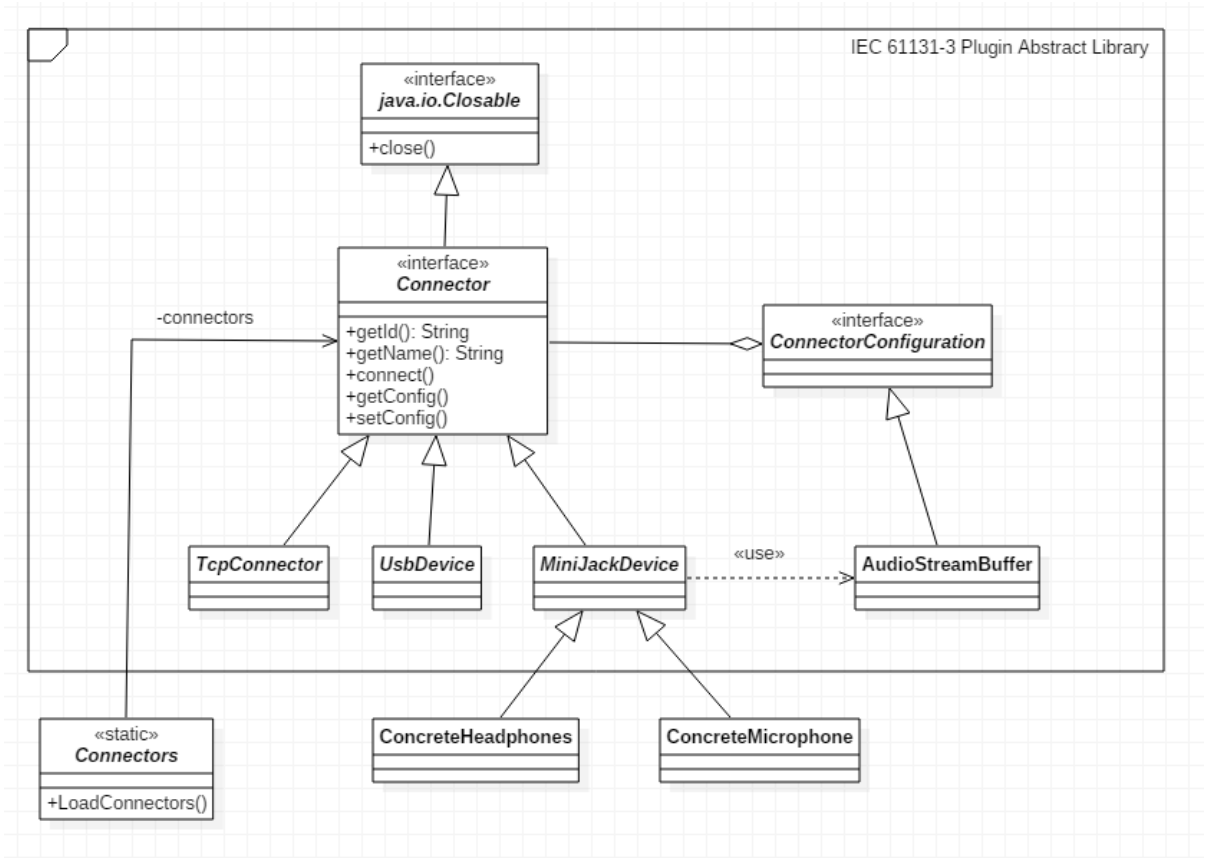

Рисунок 6. Архитектура плагина драйверов устройств.

Figure 6. Device driver plugin architecture.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

МОДУЛЬ JIT-КОМПИЛЯЦИИ

Архитектура данного модуля может быть реализована поверх того же шаблона проектирования, что и модуль интерпретатора. В проекте данного модуля потребуется описать еще одну реализацию шаблона «Посетитель» (рисунок 5), которая выполняла бы генерацию кода на низкоуровневом языке программирования Java-байткод и запускалась непосредственно из тела «InterpreterAnalysis» (рисунок 2) [11]. На более низком уровне, архитектура решения определяется возможностями сторонней библиотеки ASM, используемой для выполнения данной задачи.

ЗАКЛЮЧЕНИЕ

В рамках данной работы было проведено исследование предметной области в сфере инструментов промышленной автоматизации, исследованы различные среды выполнения для языка программирования Structured Text стандарта МЭК 61131-3 и произведен их сравнительный анализ. Была предложена и спроектирована модель трансляции языка программирования ST в режиме интерпретации на виртуальной машине Java с возможностью динамической компиляции в промежуточное представление – Java байткод.

Целью дальнейших исследований может являться анализ эффективности продукта с выбранным способом трансляции в сравнении с существующими аналогами.

СПИСОК ЛИТЕРАТУРЫ

[1] Karl-Heinz John, Michael Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems. Concepts and Programming Languages, Requirements for Programming Systems, Aids to Decision-making Tools. 2001; 376.

[2] CODESYS Control SoftPLC [Электронный ресурс] Режим доступа: https://www.codesys.com/products/codesys-runtime.html

[3] IEC 61131-3 on ECLR (The PLCnext Runtime) [Электронный ресурс] Режим доступа: http://plcnext-runtime.com/ch05-03-iec-programs.html

[4] International Standard IEC 61131-3 [Электронный ресурс] Режим доступа: https://d1.amobbs.com/bbs_upload782111/files_31/ourdev_569653.pdf

[5] Dick Grune, Ceriel J.H. Jacobs. Parsing Techniques A Practical Guide. 2003; 302.

[6] JavaCC [Электронный ресурс] Режим доступа: https://github.com/javacc/javacc

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

[7] ANTLR [Электронный ресурс] Режим доступа: https://www.antlr.org/

[8] Расширенная форма Бэкуса-Наура [Электронный ресурс] Режим доступа: https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

[9] SableCC [Электронный ресурс] – Режим доступа: https://sablecc.org/

[10] GNU Bison [Электронный ресурс] Режим доступа: https://www.gnu.org/software/bison/manual/html_node/Java-Bison-Interface.html

[11] J. W. Cooper. Java Design Patterns. A Tutorial. 2000; 257.

[12] Richard Jones, Rafael Lins., Garbage Collection. Algorithms for Automatic Dynamic Memory Management. 1996; 257.

[13] Шэнг Лиенг. Интерфейс JNI. Руководство по программированию и спецификация. 2022; 1.

REFERENCES

[1] Karl-Heinz John, Michael Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems. Concepts and Programming Languages, Requirements for Programming Systems, Aids to Decision-making Tools. 2001; 376.

[2] CODESYS Control SoftPLC [Elektronnyj resurs] Rezhim dostupa: https://www.codesys.com/products/codesys-runtime.html

[3] IEC 61131-3 on ECLR (The PLCnext Runtime) [Elektronnyj resurs] Rezhim dostupa: http://plcnext-runtime.com/ch05-03-iec-programs.html

[4] International Standard IEC 61131-3 [Elektronnyj resurs] Rezhim dostupa: https://d1.amobbs.com/bbs_upload782111/files_31/ourdev_569653.pdf

[5] Dick Grune, Ceriel J.H. Jacobs. Parsing Techniques, A Practical Guide. 2003; 302.

[6] JavaCC [Elektronnyj resurs] Rezhim dostupa: https://github.com/javacc/javacc

[7] ANTLR [Elektronnyj resurs] Rezhim dostupa: https://www.antlr.org/

[8] Rasshirennaya forma Bekusa-Naura [Elektronnyj resurs] Rezhim dostupa: https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

[9] SableCC [Elektronnyj resurs] – Rezhim dostupa: https://sablecc.org/

[10] GNU Bison [Elektronnyj resurs] Rezhim dostupa: https://www.gnu.org/software/bison/manual/html_node/Java-Bison-Interface.html

[11] J. W. Cooper. Java Design Patterns. A Tutorial. 2000; 257.

2022; 1(1)
eISSN: 2782-5280
http://oajiem.com/

[12] Richard Jones, Rafael Lins., Garbage Collection. Algorithms for Automatic Dynamic Memory Management. 1996; 257.

[13] Sh. Lieng. Interfejs JNI. Rukovodstvo po programmirovaniyu i specifikaciya. 2022; 1.

ИНФОРМАЦИЯ ОБ АВТОРАХ / INFORMATION ABOUT THE AUTHORS

Дроздов Александр Витальевич,
Сибирский Федеральный Университет, кафедра информатики института космических и информационных технологий, Красноярск, Российская Федерация
e-mail: drozdov.alexander.98@gmail.com

Alexander Vitalievich Drozdov,
Siberian Federal University, Department of Informatics, Institute of Space and Information Technologies, Krasnoyarsk, Russian Federation
e-mail: drozdov.alexander.98@gmail.com

Статья поступила в редакцию 28.06.2022; одобрена после рецензирования 18.08.2022; принята к публикации 19.08.2022.

The article was submitted 28.06.2022; approved after reviewing 18.08.2022; accepted for publication 19.08.2022.