# Титульный лист

Данная выпускная квалификационная работа посвящена разработке языка описания сценариев тестирования автомобильных систем.

# Введение

С развитием автомобильной промышленности тестирование автомобильных систем стало неотъемлемой частью жизненного цикла разработки программного и аппаратного обеспечения транспортных средств.

Современные автомобили имеют несколько блоков управления, связанных между собой внутри машинной сетью. Эти блоки взаимодействуют между собой через стандартные шинные архитектуры CAN, FlexRay, LIN и Ethernet. С развитием техники количество ECU в автомобильных системах стремительно растет, что приводит к созданию сложных структур сетей.

Для тестирования автомобильных систем, необходимо хорошо понимать, как устроен автомобиль в целом и его составляющие по отдельности. Однако представления о системе у инженеров и менеджеров зачастую очень сильно расходятся. Взаимодействие участников бизнес процесса, а именно заказчика, бизнес-аналитика, менеджера, разработчика и тестировщика, является неотъемлемой частью гибкой методологии разработки ПО

Зачастую заказчики и аналитики не имеют достаточных знаний в языках программирования общего назначения, либо использование этих языков не удобно из-за большого количества не относящихся к решаемой задаче деталей (управление памятью, низкоуровневые управляющие конструкции и так далее). Для решения этой проблемы возникла идея создать проблемно-ориентированные языки, позволяющие в области автомобильной промышленности использовать специфичные обозначения и термины. Такой язык обеспечит пользователям возможность коротко и четко сформулировать задачу и позволит уменьшить количество тестовой документации.

Основываясь на общепринятых способах проверки качества программного обеспечения, а так же специфики области автомобильной промышленности алгоритм тестирования можно описать следующим образом:

1. из тестовой инструкции извлекаются данные, необходимые для отправки на ECU;
2. данные преобразуются в кадры управления VectorBox'ом;
3. интерфейсный блок управления (VectorBox) генерирует управляющие сигналы и отправляет их на ECU;
4. после выполнения полученной инструкции ECU генерирует ответ, который отправляется на VectorBox;
5. после получения ответа от ECU VectorBox генерирует кадр и передает через интерфейс пользовательскому компьютеру (PC);
6. полученный кадр ответа транслирующая система сравнивает с ожидаемым значением, описанным в тестовой спецификации;

# Тестирование «черного ящика»

Одно и то же оборудование, разрабатываемое для разных автомобильных компаний с различной внутренней архитектурой программного обеспечения при тестировании ПО может рассматриваться в качестве «черного ящика». Данное понятие предложил У.Р. Эшби в своей книге «Введение в кибернетику». Под черным ящиком подразумевается объект исследования, внутреннее устройство которого неизвестно. В кибернетических системах оно помогает изучать поведение систем, абстрагируясь от их внутреннего устройства. Такое тестирование называется поведенческим. В этом случае проверяется функциональное поведение объекта с точки зрения внешнего мира. Под этой стратегией понимается создание тестов для тестового набора, основанных на технических требованиях и их спецификациях. Фактически, при поведенческом тестировании известно какой результат должен быть при определенном наборе данных, которые подаются на вход. Проанализировав требования и спецификации тестировщик может легко определить набор тестовых сценариев для проверки различной функциональности тестируемой системы

# BDD & DSL

* В марте 2006 года Дэн Норт предложил методологию «разработки через поведение» (behaviour-driven development, BDD), основанную на гибкой методологии разработки.

Идея: в процессе написания требования для разрабатываемого ПО аналитики должны описать тестовые сценарии таким образом, чтобы их смог понять и разработчик, и тестировщик, и заказчик.

* Почти в то же время Эрик Эванс опубликовал книгу «Предметно-ориентированное проектирование», в которой описал набор принципов и схем, направленных на создание оптимальных систем объектов.

Идея: для бизнеса удобно смоделировать систему, в которой будет определен единый язык (Domain specific language, DSL), основанный на бизнес области, такой, что бизнес словарь смогут использовать как менеджеры так и инженеры.

# Формулировка задачи

Необходимо разработать проблемно-ориентированный язык (DSL) для функционального тестирования автомобильных систем, который позволит использовать методологию «разработки через поведение» (BDD), а так же предоставит функционал, необходимый для симуляции работы автомобильных сетей.

# Аналоги

**CAPL**

Компания Vector, разрабатывающая программные инструменты для работы с коммуникационными сетями, основывающихся на шинах CAN, LIN, FlexRay, Ethernet и др., используемыми в автомобильной промышленности создала программный пакет для разработчиков: CANoe. Этот инструмент поддерживает симуляцию работы сетей, предоставляет диагностические инструменты и т.д. Данный пакет используется большинством OEM-производителей и поставщиков автомобильных компонентов. В среде CANoe есть возможность разрабатывать тестовые сценарии на языке CAPL (Communication Access Programming Language).

С помощью CAPL можно описать тесты, полностью покрывающие функциональные требования АС. Однако данный язык является Си подобным, достаточно трудным для понимания менеджерами и аналитиками.

**ССDL**

Еще одним языком для описания тестовых спецификаций в автомобильной сфере является CCDL. Это язык тестовых спецификаций для тестирования, основанном на требованиях. Предоставляет высокоуровневый язык тестирования. CCDL может использоваться для автоматизированного black box тестирования. Однако, спецификации содержат в себе программный код, и не предоставляют пользователям в свободной форме описывать тестовые шаги.

**Cucumber**

Cucumber – это фреймворк, реализующий подход BDD. В Cucumber для разработки тестов используется Gherkin-нотация. Она определяет набор ключевых слов и структуру теста. Программная реализация шагов отделена от теста, что очень удобно для бизнеса. Однако для создания полной domain specific модели АС необходимо каждый раз заново реализовывать протокол коммуникации с тестовым оборудованием. Кроме того, Cucumber не предоставляет возможность выделять блоки шагов, которые необходимо выполнить несколько раз. В тестировании АС зачастую одни и те же шаги необходимо выпонять десятки и сотни раз, а значит тестовая дкументация будет очень длинной.

**Вывод**

Из вышеперечисленных инструментов для тестирования ПО видно, что ни один из них не позволяет описывать тестовые сценарии в свободной форме, при этом предоставляя возможность проверки поведения автомобильных систем на основе работы автомобильных систем. Таким образом существует необходимость в создании предметно-ориентированного языка для функционального тестирования, который объединит в себе принципы BDD и функционал, необходимый для симуляции работы автомобильных сетей CAN, LIN, Ethernet. Наличие такого языка значительно упростит разработку ПО и повысит качество разрабатываемого продукта.

# постановка задачи

Требования к работе:

* Язык должен позволять пользователям описывать инструкции для тестовых сценариев в свободной форме.
* Синтаксис языка должен позволять описывать несколько тестовых сценариев в документе.
* Тестовые сценарии должны выполняться в той очередности, в которой описаны в документе.
* Необходимо предоставить пользователю возможность работы с коммуникационными сетями автомобильных систем.
* Результаты выполнения тестов должны сохраняться в файл.
* Необходимо реализовать обработчик ошибок.

Качество решения, представленного в данной работе должно быть проверено следующим образом:

* необходимо провести опрос среди потенциальных пользователей языка для выявления степени удобства использования синтаксиса.
* выполнить тестирование реализации языка.

Разработка языка

В качестве входов системы рассмотрим пакеты данных, отправляемые тестируемому оборудованию, а так же установку значений параметров (частота передачи данных, значения отправляемых сигналов и д.р.). В этом случае входы системы можно разделить на два класса: отправка данных и настройка данных. В таком случае выходами системы будут получаемые с тестируемого оборудования пакеты данных. Данные могут приниматься единожды (UDS пакеты), или же периодически (синхросигналы). Соответственно, представляется логичным разделить их на две большие группы: получение пакетов и данные, которые необходимо проверить в определенный момент времени. В этом случае система управления имеет входы $X(Send\_{data}, Set\_{data})$ и выходы $Y(Recieve\_{data}, Check\_{data})$.

# ANTLR4

Из вышеcказанного можно определить следующий набор ключевых слов с которых должен начинаться любой шаг сценария: {Send}, {Set}, {Check}, {Receive}, {Pause}, {Repeat}. При этом ключевые слова должны начинаться с определенного начального символа, обозначающее начало служебного слова. При тестировании программного обеспечения ECU для автомобильных систем наименее вероятно использование символа коммерческое at(@), поэтому этот символ будет обозначать ключевые слова.

Для реализации языка было выбрано «еще одно средство распознавания языков» ANTLR4 (ANother Tool for Language Recognition). Это генератор нисходящих анализаторов для формальных языков. Он преобразует контекстно-свободную грамматику в форме РБНФ в программу на Java.

На слайде представлены терминальные символы для ключевых слов Repeat и Send, а так же правило вывода тестовой спецификации.

# AST

# Реализация языка

При разработке тестовых спецификаций пользователи должна сами определить в пользовательской библиотеке метод, описывающий набор шагов, который необходимо выполнить для каждой инструкции., аннотируя метод ключевым словом и указывая текст самой инструкции. Во время обхода абстрактного синтаксического дерева необходимо по тексту инструкции, следующим после ключевого слова, находить и выполнять метод, определенный в пользовательской библиотеке. В java такая возможность может быть достигнута благодаря Reflection API.

При обходе абстрактного синтаксического дерева интерпретатор находит очередную аннотированную инструкцию. Затем помощью рефлексии находит метод, аннотированный соответствующим ключевым словом и добавляет ее в список необходимых для выполнения, все параметры пользовательских параметров приводятся к типам, объявленным в параметрах метода. Методы инструкций могут принимать параметры следующих типов: boolean, byte, short, integer, long, float, double, String, Enum.

При завершении обхода AST инструкции поочередно выполняются в том порядке, в котором были заданы в тестовой спецификации.

# XL Driver Library

Для реализации функционала симуляции автомобильных сетей используется универсальный программный интерфейс XL-Driver-Library, позволяющий получить доступ к интерфейсам аппаратных средств Vector.

Для разработки XL Driver Library приложений требуется подключить динамические библиотеки, которые находятся в открытом доступе на сайте компании Vector. Необходимые методы реализованы на языке С, поэтому для доступа к методам через java был реализован класс JNIVxlApi.java c использованием механизма JNI.

Java Native Interface (JNI) — стандартный механизм для запуска кода, под управлением виртуальной машины Java (JVM), который написан на языках С/С++ или Ассемблера, и скомпонован в виде динамических библиотек, позволяет не использовать статическое связывание. Это даёт возможность вызова функции С/С++ из программы на Java, и наоборот.

Результаты выполнения тестовых сценариев записываются в текстовый файл results.txt.

# Тестирование интерпретатора

Для тестирования реализованного языка были разработаны Unit тесты. Модульное тестирование, оно же юнит-тестирование, позволяет проверить корректность отдельных модулей исходного кода программы. Для этого было реализовано восемь junit-тестов с использованием пакета org.junit. Были проверены следующие тестовые случаи:

1. в качестве параметра для паузы указано не целочисленное значение, а текст;
2. значение для паузы не указано;
3. значение для паузы указано верно, в следующем шаге значение для паузы не указано;
4. не указана текстовая инструкция после ключевого слова @Send;
5. не указана текстовая инструкция после ключевого слова @Send, далее указан корректный шаг;
6. не указана текстовая инструкция после ключевого слова @Set, далее указан корректный шаг
7. указаны две корректные инструкции на одной и той же строке
8. указан шаг, тестовая инструкция для которого не определена в тестовой библиотеке;

# Описание полученных результатов

В качестве примера применения разработанного языка представлен пример тестового сценария для проверки функционала записи данных в энергонезависимую память. В качестве тестируемого оборудования использовался лидар.

Результаты теста, записанные в файл result.txt представлены на слайде. Из данного файла видно, что ожидаемый ответ от тестируемого оборудования о том, что данные успешно записаны (62 F1 01) был получен. Однако после перезапуска тестируемого оборудования ожидаемый ответ (62 F1 01 00 01 A5) на запрос о прочтении данных не был получен. Вместо этого тестируемое оборудование отправило ответ (7F 22 31), означающий, что данный сервис не поддерживается в текущей сессии.

# Заключение

В результате проведенной работы был разработан и реализован язык описания сценариев тестирование программного обеспечения автомобильных систем.

Язык позволяет пользователям описывать инструкции для тестовых сценариев в свободной форме.

В одном документе можно описать несколько тестовых сценариев, выполняющихся поочередно в той последовательности, в которой были указаны в тестовой спецификации. Данные, полученные в результате выполнения теста сохраняются в отдельный файл.

По результатам опроса, проведенного среди опытных тестировщиков автомобильных систем выявлено, что разработанный язык мог бы использоваться на производстве. Однако очевидно, язык может быть улучшен. Для удобства использования разработанного языка в дальнейшем стоит изменить формат вывода результатов и представлять их в качестве таблицы, или же в каком-либо другом формате. Так же стоит реализовать графический интерфейс для удобства пользователей, в котором в режиме реального времени могли бы выводиться результаты тестового прогона.