Содержание

[Введение 6](#_Toc327704001)

[1 Прототипирование архитектуры ПО 7](#_Toc327704002)

[1.1 Прототипирование ПО 7](#_Toc327704003)

[1.2 Понятие прототипа архитектуры ПО 10](#_Toc327704004)

[1.3 Использование прототипов архитектуры ПО 11](#_Toc327704005)

[1.4 Требования к прототипам архитектуры ПО 16](#_Toc327704006)

[1.5 Инструменты прототипирования архитектуры ПО 17](#_Toc327704007)

[1.6 Проверка правильности прототипов 18](#_Toc327704008)

[1.7 Выводы 19](#_Toc327704009)

[2 Модель системы прототипирования 20](#_Toc327704010)

[2.1 Общие положения 20](#_Toc327704011)

[2.2 Модель прототипа архитектуры 21](#_Toc327704012)

[2.3 Описание модели прототипа 22](#_Toc327704013)

[2.3.1 Обзор языков автоматного программирования 22](#_Toc327704014)

[2.3.1.1 AsmL 23](#_Toc327704015)

[2.3.1.2 FSMGenerator 23](#_Toc327704016)

[2.3.1.3 SMC (State Machine Compiler) 23](#_Toc327704017)

[2.3.1.4 State Machine 24](#_Toc327704018)

[2.3.1.5 ТАВР 24](#_Toc327704019)

[2.3.1.6 Ruby с использованием библиотеки STROBE 25](#_Toc327704020)

[2.3.1.7 JetBrains Meta Programming System (MPS) 25](#_Toc327704021)

[2.3.1.8 UniMod 26](#_Toc327704022)

[2.3.2 Требования к языку описания модели прототипа 26](#_Toc327704023)

[2.4 Верификация модели прототипа 27](#_Toc327704024)

[2.5 Model Checking 29](#_Toc327704025)

[2.6 Верификатор SPIN 30](#_Toc327704026)

[2.6.1 Язык Promela 31](#_Toc327704027)

[2.6.2 Модель Крипке 32](#_Toc327704028)

[2.6.3 Язык темпоральной логики LTL 34](#_Toc327704029)

[2.6.4 Автомат Бюхи 35](#_Toc327704030)

[2.7 Выводы 36](#_Toc327704031)

[3 Реализация системы 38](#_Toc327704032)

[3.1 Общее описание 38](#_Toc327704033)

[3.2 Подсистема лексического и синтаксического анализа 39](#_Toc327704034)

[3.2.1 Структура подсистемы лексического и синтаксического анализа 39](#_Toc327704035)

[3.2.2 Описание языка Proto 39](#_Toc327704036)

[3.2.2.1 Синтаксис языка 40](#_Toc327704037)

[3.2.2.2 Грамматика языка 41](#_Toc327704038)

[3.2.2.2 Пример программы на языке proto 43](#_Toc327704039)

[3.2.3 Выбор средств реализации 46](#_Toc327704040)

[3.2.4 Расширенная форма Бэкуса-Наура 48](#_Toc327704041)

[3.2.5 Структура абстрактного синтаксического дерева 48](#_Toc327704042)

[3.3 Подсистема построения графа потока исполнения 50](#_Toc327704043)

[3.3.1 Общее описание 50](#_Toc327704044)

[3.3.2 Алгоритм построения графа потока исполнения по абстрактному синтаксическому дереву программы 50](#_Toc327704045)

[3.4 Подсистема кодогенерации 52](#_Toc327704046)

[3.4.1 Построение модели на языке Promela 53](#_Toc327704047)

[3.4.2 Преобразование LTL-формул 54](#_Toc327704048)

[3.4.3 Типовые требования к автоматным моделям 56](#_Toc327704049)

[3.5 Plug-in для Eclipse 57](#_Toc327704050)

[Заключение 58](#_Toc327704051)

[Список использованных источников 59](#_Toc327704052)

# Введение

Современные методы и практики быстрой разработки программного обеспечения нацелены на минимизацию рисков, путём сведения процесса к серии коротких циклов, каждый их которых представляет программный проект в миниатюре и включает все задачи, необходимые для выдачи минимального прироста по функциональности. При таком подходе особенно важно представлять всю систему целиком на ранних этапах разработки. Для решения подобных задач применяются прототипы программных систем, которые позволяют вести разработку проекта методом непрерывной интеграции.

Настоящая работа представляет собой исследование современных решений в области прототипирования программного обеспечения, оценку их эффективности и применимости согласно выдвинутой модели требований, а также выводы о необходимости появления нового класса инструментов прототипирования, в виду неготовности существующих решений удовлетворять ранее выдвинутым требованиям.

Кроме того, в работе детально представлена предлагаемая автором модель системы описания и тестирования прототипов архитектуры программного обеспечения и рассмотрена ее реализация с точки зрения современных технологий программирования. Основополагающая идея предлагаемого подхода заключается в использовании унифицированного языка в процессе описания прототипов, а также свойств конечных автоматов в процессе тестирования.

# 1 Прототипирование архитектуры ПО

## 1.1 Прототипирование ПО

Конструирование или проектирование программного обеспечения – это непростой процесс, требующий много усилий и внимания. Хорошо спроектированное приложение может в дальнейшем сэкономить много времени и сил программисту. Проектирование является едва ли не самой крупной частью процесса разработки и в значительной степени влияет на успешность всего процесса в целом. Кроме того проектирование является своего рода центральной частью всего процесса работы над программой: до него происходит этап составления требований к программному обеспечению и начального построения его архитектуры, после него – тестирование и устранение ошибок приложения.

Результатом этапа конструирования является готовый исходный код программы. Спецификации требовании и проектная документация могут устареть, но исходный код актуален всегда, и именно поэтому он должен быть максимально качественным. Часто процесс конструирования также называют «кодированием» или «программированием». Конструирование — единственный процесс, который выполняется во всех случаях [1, 2].

Идеальный программный проект до начала конструирования проходит стадии тщательной выработки требований и проектирования архитектуры [3]. После конструирования в идеале должно быть выполнено исчерпывающее тестирование системы. Однако в реальных проектах разработчики часто пропускают этапы выработки требований и проектирования, начиная прямо с конструирования программы. Тестирование также часто выпадает из расписания из-за огромного числа ошибок и недостатка времени. Отсюда можно сделать вывод, что повышение эффективности конструирования программного обеспечения позволяет оптимизировать любой проект, каким бы несовершенным он ни был. Кроме того качество конструирования ПО - это один из основополагающих факторов оценки качества всего ПО в целом.

В процессе конструирования условно можно выделить следующие этапы:

* проектирование – выработка требований, продумывание архитектуры приложения, отрисовка диаграмм (интерфейсов, классов и т.д.),
* кодирование – написание программного кода в соответствии с результатами, полученными на этапе детального проектирования,
* отладка – этап конструирования, на котором локализуются и устраняются ошибки в коде программы,
* интеграция – процесс объединения отдельных компонентов в единую систему,
* тестирование – процесс выявления ошибок программы с применением различных подходов. Он позволяет получить представление о качестве спроектированного приложения и исправить ошибки, которые не были выявлены и исправлены на этапе отладки.

Если в результате этапа конструирования мы получаем готовый программный продукт, то процесс прототипирования [4, 5] позволяет в результате получить макет готовой системы. Такой макет можно проверить на пригодность предлагаемых для применения концепции, архитектурных и технологических решений, а так же предоставить на ранних этапах разработки заказчику.

Прототипирование можно назвать ускоренной версией проектирования, поскольку в этом случае к основным стадиям можно отнести следующие:

* определение начальных требований,
* разработка первого варианта прототипа системы на основе требовании,
* изучение прототипа и получение обратной связи о необходимых изменениях и дополнениях,
* переработка и улучшение прототипа (с учётом полученных замечаний и предложений изменяются как спецификации, так и прототип).

Зачастую результат, полученный в процессе прототипирования, может и не стать частью готовой системы. Но, тем не менее, он может служить ещё одним шагом на пути к созданию прототипа финальной версии разрабатываемого продукта.

Выделяют различные подходы к прототипированию. Но, в общем, принято выделять два основных, принципиально различных подхода – это быстрое и эволюционное прототипирование. Эволюционное прототипирование заключается в последовательном создании макетов системы, которые будут все ближе и ближе к реальному продукту. Несомненное преимущество такого подхода в том, что на каждой шаге мы имеем рабочую систему, пусть и не располагающую всеми необходимыми нам функциями, но уже более приближенную к финальной версии, нежели предыдущая версия системы. Такой подход очень удобен в ситуации, когда все требования к системе ещё не определены, и будут определяться в процессе разработки. Однако при использовании быстрого прототипирования заранее предполагается, что создаваемый макет на каком-то этапе будет оставлен и не войдёт в готовую систему. Безусловно, преимуществом такого подхода является скорость – в ответ на требования заказчика сразу проектируется каркас системы. Этот каркас системы отдаётся заказчику, требования вновь уточняются или изменяются и вновь происходит создание каркаса. Стоимость внесения изменений и создания нового каркаса очень низкая, поскольку на этом этапе не нужно писать код системы, а создаётся только её каркас. К преимуществам использования прототипирования как такового можно отнести уменьшение времени разработки и стоимости системы за счёт улучшения спецификации, а также вовлечение пользователей или заказчиков в процесс разработки.

Тем не менее, у прототипирования так же можно выделить следующие недостатки:

* недостаточный анализ (акцентирование внимания разработчиков на ограниченном прототипе может отвлечь их от анализа требовании на итоговую систему),
* чрезмерное время на создание прототипа (если разработчики проектируют слишком сложную систему и тратят много времени, то все преимущества от использования прототипирования теряются),
* смешение представлений пользователей или заказчиков о прототипе и готовой системе (есть вероятность, что они могут потерять отличие между прототипом и основой будущей системы и разочароваться в возможностях разработчиков).

На этапе работы над каркасом приложения удобно пользоваться различными вспомогательными средствами, среди которых можно выделить универсальный язык графического описания для объектного моделирования UML [6]. С его помощью можно получить прототип системы в виде различных схем и диаграмм, которые с разных сторон отразят особенности разрабатываемой архитектуры приложения. Полученные диаграммы проверяются только на правильность и соответствие стандартам UML и не гарантируют правильность относительно требований, предъявленных к разрабатываемой системе.

Можно создавать прототипы следующих типов:

* прототип архитектуры,
* прототип новой функциональной возможности уже существующей системы,
* прототип структуры или содержания внешних данных системы,
* прототип инструментальных средств или компонентов,
* прототип рабочих характеристик,
* прототип дизайна пользовательского интерфейса.

## 1.2 Понятие прототипа архитектуры ПО

Прототип архитектуры программного обеспечения [7] – это упрощённое представление желаемой архитектуры, без учёта деталей и аспектов реализации. Прототипы архитектуры создаются, чтобы смоделировать будущую систему в целом. Ни один из отдельных модулей в прототипе не должен быть особенно функциональным. Основная цель создания прототипа архитектуры в том, чтобы смоделировать разрабатываемую систему и понять, как она будет выглядеть в собранном виде, опуская детали.

## 1.3 Использование прототипов архитектуры ПО

Когда разработчики сталкиваются с разработкой чего-то нового и ещё не существующего, они в первую очередь подвержены большому риску выбрать неверный способ и пойти по неправильному пути. А поскольку заказчики и пользователи ранее не сталкивались с подобного типа системами, то и требования их могут быть неточными и расплывчатыми. Кроме того, сами разработчики вынуждены будут использовать средства алгоритмы, методики или библиотеки, с которыми они не знакомы. Таким образом, получается, что разработчики сталкиваются с большим количеством неизвестных.

Самый часто встречающийся выход из этой ситуации – это составление предельно подробных спецификаций системы. Написание большого количества документации, которые будут чётко регламентировать каждое требование к системе, связывать каждое неизвестное и ограничивать рабочую среду.

Однако существуют и другие способы решения такой проблемы, одним из которых является «стрельба трассирующими» [8]. Характерной особенностью данного метода является то, что на начальном этапе разрабатывается так называемый «скелет» системы, который состоит лишь из базовых элементов системы, без деталей и особенностей. Затем к этому «скелету» постепенно дополняется новые функциональные возможности путём параллельного наращивания каждого компонента. Этот «скелет» также принято называть программой трассировки. Она содержит всю проверку ошибок, документацию и структурирование, которые имеются в любом фрагменте рабочей программы. Единственное её отличие в том, что она не обладает всеми функциональными возможностями. В то же время, как только разработчики смогут добиться сквозного соединения между компонентами системы, то смогут оценить, насколько близко они находятся к цели, и в случае необходимости сделать поправку. Как только разработчики попадают в цель – добавление функциональных возможностей значительно облегчается.

Разработка программы трассировки идёт в согласии с той идеей, что работа над проектом никогда не заканчивается, то есть всегда будет потребность в добавлении нового функционала и потребность в изменениях. Такой подход к разработке называется инкрементальным.

Альтернативой этому подходу является тяжеловесный технический подход, при котором вся разрабатываемая система делится на модули, разработка которых ведётся в вакууме. Модули объединены в подсистемы, которые в дальнейшем тоже подлежат объединению, пока в конечном итоге не получится завершённое приложение. Оно то и может быть представлено конечному пользователю и протестировано.

Технология программы трассировки имеет следующие преимущества:

* *можно предоставить пользователям некий работающий вариант системы ещё до выпуска окончательной версии*. Если пользователи будут осознавать, что видят перед собой не конечную версию продукта, а нечто промежуточное, то с их помощью можно будет отследить, насколько близко к цели находится та или иная итерация;
* *разработчики выстраивают некую структуру, в которой они в дальнейшем работают*. Как известно, наибольший страх вызывает лист бумаги, на котором ничего не написано. Если уже разработаны механизмы взаимодействия между модулями системы и есть их реализация, то команде разработчиков не придётся много выдумывать. Этот факт делает труд каждого члена команды более производительным и, безусловно, способствует последовательности в работе;
* *есть платформа для интеграции.* Как только все компоненты системы будут связаны друг с другом, появится некая среда, в которую уже можно будет добавлять новые фрагменты программ, которые пройдут модульное тестирование. Впоследствии, необходимо будет заниматься интеграцией каждый день, а то и по нескольку раз. При каждой интеграции в среду будет добавляться небольшой фрагмент, и не будет происходить «большого скачка». А значит и воздействие каждого нового изменения становится более очевидным, взаимодействия более ограниченными, вследствие чего отладка и тестирование будут проходить быстрее и точнее;
* *есть что продемонстрировать.* В случае неожиданного запроса со стороны заказчиков увидеть демонстрационные версии системы всегда будет, что им продемонстрировать;
* *лучше ощущается прогресс.* При разработке программы трассировки программисты всегда работают над сценариями использования системы в соответствии с очередью. Когда они заканчивают работу над одним сценарием, то переходят к другому. При таком подходе к работе гораздо проще контролировать производительность и показать заказчику продвижение в проекте. А так как каждая индивидуальная разработка по объёму очень мала, то можно избежать создания больших монолитных программных блоков;

Как известно, трассирующие пули только показывают, куда произошло попадание. Но не обязательно это попадание было в цель. С учётом того, куда вы попали, вы затем корректируете прицел, пока не добьётесь попадания в цель. То же самое и относится к программе трассировки. Эту методику используют в ситуациях, когда нет уверенности на 100% в том, куда дальше двигаться. Не стоит удивляться возникновению ситуации, что программа трассировки работает совсем не так как того хочет пользователь или, например, возникают проблемы с производительностью. Просто необходимо выработать подход для изменения того, что мешает приблизиться к цели. Небольшой фрагмент программы обладает малой инерцией – его можно легко и быстро изменить.

Принцип программы трассировки в какой-то степени перекликается с гибкой методологией разработки (*Agile Software Development*) [9, 10, 11]. Такие методологии разработки предполагают сведение разработки программного обеспечения к серии коротких циклов с целью минимизации рисков. Такие короткие циклы ещё называют итерациями. Продолжительность их может быть 2-3 недели. Каждая итерация выглядит как программный проект в миниатюре и включает все задачи, необходимые для выдачи мини-прироста по функциональности: планирование, анализ требовании, проектирование, кодирование, тестирование и документирование. Несмотря на то, что отдельная итерация недостаточна для выпуска новой версии продукта, считается, что гибкий программный продукт готов к выпуску в конце каждой итерации. По окончанию каждой итерации необходимо проводить переоценку приоритетов и затем приступать к новой итерации.

Также одной из основных особенностей этих методов (так называемых Agile-методов) является то, что упор при работе делается на непосредственное общение лицом к лицу. При таком подходе уменьшается объем письменной документации по сравнению с другими методами. Основной метрикой agile-методов является программный продукт.

Agile представляет собой не единственный подход к организации процесса разработки программного обеспечения, а целое семейство процессов разработки, которое определяется положениями специально изданного манифеста Agile Manifesto [12]. Среди методологии разработки, которые придерживаются манифеста можно выделить следующие методологий:

* *Agile Modeling* - набор понятий и приёмов, позволяющих быстро и просто выполнять моделирование и документирование в проектах разработки программного обеспечения. Основная цель – это эффективное моделирование и документирование; но не охватывает программирование и тестирование, не включает вопросы управления проектом, развёртывания и сопровождения системы. Однако включает в себя проверку модели кодом;
* *Agile Unified Process (AUP)* упрощённая версия IBM Rational Unified Process (RUP), которая описывает простое и понятное приближение (модель) для создания программного обеспечения для бизнес-приложений;
* *Agile Data Method* - группа итеративных методов разработки программного обеспечения, в которых требования и решения достигаются в рамках сотрудничества разных кросс-функциональных команд;
* *DSDM* основан на концепции быстрой разработки приложений (Rapid Application Development, RAD). Представляет собой итеративный и инкрементный подход, который придаёт особое значение продолжительному участию в процессе пользователя/потребителя;
* *Open Unified Process*(OpenUP) - это итеративно-инкрементальный метод разработки ПО. Делит жизненный цикл проекта на четыре фазы: начальная фаза, фазы уточнения, конструирования и передачи;
* *Getting Real* - это итеративный подход без функциональных спецификаций, использующийся для веб-приложений. В данном методе сначала разрабатывается интерфейс программы, а потом её функциональная часть;
* *Extreme Programming* [13] – процесс разработки, основанный на 12 основных приёмах, которые объединены в 4 группы: короткий цикл обратной связи, непрерывный процесс разработки, разделяемое всеми понимание и социальная защищённость программиста;
* *Scrum* [14] - это набор принципов, на которых строится процесс разработки, позволяющий в жёстко фиксированные небольшие промежутки времени (2-4 недели) предоставлять конечному пользователю работающее ПО с новыми возможностями, для которых определён наибольший приоритет. Использование этой методологии даёт возможность выявлять и устранять отклонения от желаемого результата на более ранних этапах разработки программного продукта;
* *а также многие другие*.

Можно было подумать, что все вышеперечисленные подходы к организации процесса разработки программного обеспечения есть не что иное, как разработка прототипа системы. Но есть большое отличие между созданием прототипа системы и любым из этих методов. Цель работы над прототипом – это исследование определённых характеристик и аспектов конечной версии системы. При создании истинного прототипа будет отброшено все то, что критиковалось при тестировании прототипа, и он будет переписан с учётом полученных фактов. А, например, в подходе типа «стрельба трассирующими» есть необходимость знать о том, как приложение работает в целом. Разработчикам нужно дать пользователям некий «скелет» архитектуры, на который в дальнейшем будет происходить наращивание тела программы. Прототипы генерируют одноразовую программу, в отличие от программ, полученных одним из вышеуказанных методов – простых, но завершённых, и образующих часть каркаса конечной версии системы. Лучше рассматривать создание прототипа как сбор сведений разведки до начала стрельбы.

Таким образом, прототипирование – важная начальная стадия многочисленных методов и принципов разработки программных продуктов, позволяющая значительно минимизировать риски при разработке, а так же избежать большого количества ошибок.

## 1.4 Требования к прототипам архитектуры ПО

Цель работы с прототипом – исследование определённых характеристик и аспектов конечной версии системы. При построении прототипа можно пренебречь деталями и особенностями системы, которые в данный момент не важны.

Детали, которые можно не учитывать при работе над прототипом архитектуры программной системы:

* корректность – там, где это приемлемо, можно использовать фиктивные данные,
* завершённость – прототип может функционировать лишь в ограниченном смысле, возможно лишь с одним заданным фрагментом данных и одним пунктом меню,
* надёжность – процедура проверки ошибок, вероятно, будет неполной или будет отсутствовать полностью. При отклонении от определённого пути прототип может выйти из строя,
* стиль – прототип программы не имеет большого значения для комментариев или документации.

Готовый прототип архитектуры системы может помочь получить ответы на многие вопросы. Например:

* чётко ли определены обязанности основных компонентов,
* являются ли эти обязанности приемлемыми для компонентов,
* чётко ли определена совместная работа основных компонентов,
* сведено ли к минимуму связывание между компонентами,
* можно ли выделить потенциальные источники дублирования,
* можно ли применять определения интерфейсов и ограничения,
* обладает ли каждый из модулей путём доступа к данным, требуемым ему в ходе выполнения? Может ли он получить его в случае необходимости.

## 1.5 Инструменты прототипирования архитектуры ПО

Поскольку большинство прототипов создаётся с целью моделирования рассматриваемой системы в целом, то полученный прототип есть не что иное, как одноразовая программа, необходимая для того, чтобы получить ответы на определённый ряд вопросов. В прототипах опущены ненужные детали и подробности, что позволяет в центре рассмотрения иметь лишь определённые аспекты системы. Для создания прототипа не нужно писать программу – он может быть составлен даже на обычном листе бумаги или доске. Главная цель составления такого прототипа – это получить понимание того, как система будет выглядеть в собранном виде, опуская детали. С этой точки зрения может показаться удобным создание прототипов посредством языков очень высокого уровня, а точнее языков более высокого уровня по сравнению с языком, используемым при написании системы. К таким языкам, например, можно отнести Perl [15] и Python. Язык сценариев высокого уровня позволяет опустить многие детали (например, указание типов данных), но при этом создавать функциональный фрагмент программы. Такие языки также позволят при необходимости соединить низкоуровневые фрагменты в новые сочетания. В итоге, используя такой подход, можно быстро собрать существующие компоненты в новые конфигурации и посмотреть, как они работают.

## 1.6 Проверка правильности прототипов

Правильность – это одно из самых важных свойств программного обеспечения, а методы её проверки и обеспечения – это, несомненно, очень важная и актуальная проблема, требующая исследований.

На сегодняшний день можно выделить два подхода к проверке правильности прототипов: динамический и статический [16].

Динамический подход заключается в проведении проверки непосредственно в процессе исполнения. Самый простой и часто встречающийся пример этого подхода – это тестирование.

Статический подход основывается на проверке правильности системы в том или ином смысле. Простейший вид проверки можно встретить в любом компиляторе (например, проверка синтаксиса программного текста, валидация). Одним из наиболее мощных видов данного подхода является формальная верификация [17] – доказательство соответствия текста программной системы её формальной спецификации.

В формальной верификации выделяют несколько основных направлении:

* *доказательная верификация* [18, 19]. Требует большого объёма ручной работы, что приводит к неприменимости для больших программных систем;
* *проверка модели* системы *(Model Checking)* [20]. Данный подход к верификации требует предварительной подготовки: построения по программе некой модели с конечным числом состояний, описания требований к программе в терминах одного из видов темпоральной логики [21]. Результатом верификации модели может быть подтверждение того, что модель удовлетворяет предъявленным требованиям, либо контрпример. Построение контрпримера требует выявления причины некорректности (например, наличие ошибок в исходной программе, построенной модели или спецификации). Решение этих задач плохо поддаётся автоматизации для программ общего вида.

## 1.7 Выводы

Согласно приведённым выше рассуждениям, можно сделать вывод о том, что основополагающая проблема прототипирования архитектур программных проектов заключается в отсутствии на рынке целого класса специализированных систем, комплексно удовлетворяющих выдвинутым требованиям. Кроме того, современные тенденции развития методологий и практик в области быстрой интегральной разработки программного обеспечения, лишь подтверждают необходимость в появлении подобных инструментов прототипирования.

Таким образом, можно сделать вывод о необходимости создания системы, удовлетворяющей рассмотренным требованиям. Автором спроектирован и разработан проект системы описания и тестирования прототипов программных архитектур, позволяющей обеспечить выполнение перечисленных требований. Основополагающая идея предлагаемого автором подхода заключается в использовании унифицированного языка в процессе описания прототипов, а также свойств конечных автоматов в процессе тестирования.

# 2 Модель системы прототипирования

## 2.1 Общие положения

Автором предлагается модель архитектуры системы прототипирования, которая позволяет обеспечить выполнение рассмотренных в первом разделе требований. Главная идея предлагаемого подхода состоит в использовании унифицированного языка в процессе описания прототипов, а также свойств конечных автоматов в процессе тестирования.



Рисунок 2.1 – Описание работы системы

В работе системы прототипирования можно выделить несколько этапов (см. Рисунок 2.1):

* построение описания прототипа системы;
* получение модели системы на автоматном языке;
* запуск верификатора и получение программы на языке высокого уровня;
* компилирование исполняемой программы;
* в случае ошибки – поиск пути с ошибкой;
* построение контрпримера.

## 2.2 Модель прототипа архитектуры

Под прототипом архитектуры будем далее понимать минимальный набор сущностей предметной области, который позволяет с одной стороны осветить проектируемую архитектуру со всех сторон, но в то же время не обременить её избыточными деталями.

Под моделью прототипа архитектуры далее будем понимать абстракцию, которую можно охарактеризовать с помощью кортежа вида:

*M = (S, s0, F, R, U)*, где

S – конечное множество состояний прототипа,

s0 – начальное состояние, s0 S,

F – множество конечных состоянии, F ⊆ S,

R – набор спецификаций требований прототипа,

U – множество сценариев использования прототипа.

Верификация модели прототипа – это основной метод поиска ошибок в прототипе архитектуры. Модель для процесса верификации строится на основе прототипа путём выделения состоянии, сбора спецификации требовании по работе прототипа, а также формализации всех возможных сценариев использования прототипа.

Состояние в работе прототипа можно определить как момент в работе прототипа архитектуры, который объединяет в неявной форме все входные воздействия прошлого, а также влияет на реакцию в текущий момент времени. А значит, реакция прототипа зависит от полученного входного воздействия и состояния, в котором в данный момент находится прототип. Каждое состояние имеет вполне определённый смысл и качественно отличается от всех других состоянии, и кроме того однозначно определяет действия, которые могут совершаться в этом состоянии.

Набор спецификаций модели прототипа представляет собой список свойств, которыми должен обладать прототип в том или ином состоянии. Иными словами, спецификации требовании отражают желаемое поведение прототипа архитектуры. Одним из способов, который позволяет специфицировать требования к прототипу, является использование исчислений темпоральной логики [23]. Темпоральная логика представляет собой исчисление, специализированное для учёта причинно-следственных связей во времени. С помощью такой логики можно легко установить последовательность и взаимосвязь явлений по временной шкале. Свойства систем описываются в темпоральной логике при помощи темпоральных формул.

Сценарий прототипа представляет собой множество правил перехода между состояниями, каждое из которых отражает возможность выполнения каждого требования спецификации. Главная задача сценария состоит в том, чтобы подтвердить или опровергнуть выполнение какого-либо свойства из набора спецификации. Сценарии, которые указывают на невозможность выполнения спецификации, будем называть контрпримерами.

## 2.3 Описание модели прототипа

При описании модели прототипа архитектуры удобно использовать унифицированный язык. Такой язык должен поддерживать все установленные правила описания и позволять проводить описание любой архитектуры системы.

Поскольку в основе модели прототипа лежит конечный автомат, то есть вероятность, что описание такой системы удобно будет провести с помощью языка автоматного программирования.

### 2.3.1 Обзор языков автоматного программирования

На данный момент существует небольшое количество автоматных языков. Каждый из таких языков имеет свои особенности, поскольку разрабатывался в соответствии с определённой целью. Рассмотрим существующие языки автоматного программирования, выделим их особенностей, а затем сформулируем требования к разрабатываемому языку.

#### 2.3.1.1 AsmL

AsmL (Abstract State Machine Language) – это язык, разработка которого ведётся компанией Microsoft. Этот язык предназначен в первую очередь для спецификации функциональности компьютерных систем [24] и базируется на понятии абстрактного состояния, в роли которого можно рассматривать вектор значений всех переменных. Таким образом, переход в новое состояние осуществляется при изменении переменных, поэтому синтаксис языка не предусматривает явного задания графа переходов автомата. Данный язык разработан для платформы .NET с возможностью интеграции с другими языками программирования для этой платформы.

#### 2.3.1.2 FSMGenerator

FSMGenerator (Finite State Machine Generator) представляет собой язык описания шаблонов конечных автоматов, а также средство трансляции этих шаблонов в код на языках программирования C++ или Java [25]. В шаблоне должны быть заданы свойства и сущности автомата, например, имя автомата, множества состояний, событий, переходов и действий. Такой способ задания шаблона приводит к излишней избыточности и неудобству использования FSMGenerator в качестве языка программирования. Таким образом, рассматриваемый подход достаточно удобен лишь для хранения данных автомата.

#### 2.3.1.3 SMC (State Machine Compiler)

SMC (State Machine Compiler) – язык автоматного программирования, в котором в отличие от языка *AsmL* явно описывается граф переходов автомата [26]. Описание состояния автомата происходит с помощью отдельных блоков, которые содержат правила переходов в соответствии с событиями автомата. Несомненное преимущество программ на данном языке – это возможность их трансляции в код на одном из распространённых языков программирования. Небольшое ограничение в использование этого языка вносит невозможность использования вложенных автоматов. А так же можно отметить бедность синтаксиса, что в свою очередь приводит к снижению читаемости кода программ.

#### 2.3.1.4 State Machine

Данный язык является расширением языка программирования Java за счёт введения в него автоматных конструкций [27]. В основе этого языка лежит одноименный паттерн проектирования State. Реализация каждого состояния происходит с помощью добавления отдельного класса, что в свою очередь позволяет сделать состояния максимально независимыми друг от друга. Однако в тоже время это приводит к избыточности кода, а также увеличению числа и размера классов. Таким образом, язык State Machine с одной стороны упрощает использование соответствующего паттерна, но с другой стороны практически исключает возможность эффективного описания сложных автоматов.

#### 2.3.1.5 ТАВР

ТАВР (Textual Automata Based Programming) – это язык, который сочетает в себе некоторые положительные качества всех описанных выше языков [28]. Данный язык даёт возможность неявного задания графа переходов, задания нескольких автоматов в рамках одной программы, а также наследования, параметризации и обусловливания событий. Кроме того, можно считать, что этот язык выходит за рамки предметно-ориентированного и является языком общего назначения. Такая универсальность языка автоматного программирования неизбежно оказала влияние на его сложность. Будучи универсальным, язык TABP не требует интеграции с другими универсальными языками, однако с помощью интерфейсов может быть осуществлена связь с системой и другими приложениями.

#### 2.3.1.6 Ruby с использованием библиотеки STROBE

Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby с использованием библиотеки STROBE [29] помогает решить проблему переноса диаграмм переходов автоматов, разработанных по SWITCH-технологии [30, 31], в исполняемый код. Среди особенностей этого подхода можно выделить декларативную структуру кода, а также изоморфность кода исходной диаграмме. Данный язык допускает наличие нескольких экземпляров одного и того же автомата, обеспечение связей между ними, а также интеграцию с программами на других языках и возможность управления физическими устройствами. Тем не менее, использование интерпретируемого языка Ruby [32] в качестве базового языка накладывает определённые ограничения на сферу применимости данного подхода, среди которых можно выделить отсутствие поддержки потоков операционной системы, а также отсутствие компиляции в байткод и встроенной поддержки юникода. Правда, последние два ограничения обходятся использованием специальных компиляторов для компиляции в Java и .NET байткоды и использованием дополнительных библиотек. Однако это повлияет на скорость работы программы, которая и без того для этого подхода будет невысока. Таким образом, основной недостаток этого подхода – привязанность к языку программирования Ruby.

#### 2.3.1.7 JetBrains Meta Programming System (MPS)

Без сомнения заслуживают внимания языки автоматного программирования, созданные с помощью системы метапрограммирования JetBrains Meta Programming System (MPS) [33]. Данная система делает возможным создание предметно-ориентированных языков путём задания определённых моделей и редакторов для языка. Структура и внешний вид этих редакторов таковы, что пользователь, работая с текстом программы, непосредственно напрямую редактирует абстрактное синтаксическое дерево [34] программы, а, следовательно, и модель конечного автомата в случае языка автоматного программирования. Это выводит языки рассматриваемой системы из категории текстовых языков. С помощью системы MPS созданы два языка автоматного программирования: первый – в виде самостоятельного языка, второй – в виде расширения языка Java. Эти языки позволяют описывать состояния и логику переходов автоматов по событиям, а также сами события. Кроме того, обеспечена возможность автоматического построения диаграммы состояний по мере набора текста. Заметим, что для рассмотренных выше языков не представляется подобная возможность. Таким образом, можно выделить два недостатка данного подхода: привязанность к системе MPS (до трансляции в язык общего назначения), доступность диаграмм состояний только в режиме просмотра.

#### 2.3.1.8 UniMod

Проект UniMod [35] предлагает визуальный язык автоматного программирования, поддерживая концепцию «Исполняемый UML» [36]. С помощью данного языка строятся два типа UML-диаграмм: диаграммы классов и диаграммы состояний [37]. Диаграмма классов изображает автоматы, поставщики событий, объекты управления и связи между ними. При этом автоматы описываются диаграммами состояний, а поставщики событий и объекты управления – классами на языке Java. Применение этого подхода для реализации систем со сложным поведением показало его эффективность, но также выявило недостаток – трудоёмкость визуального ввода диаграмм.

### 2.3.2 Требования к языку описания модели прототипа

Выделим требования к разрабатываемому языку автоматного программирования с учётом рассмотренных выше языков.

Одним из важнейших требований является возможность явного задания графа переходов. Это позволит гарантировать изоморфность заданного графа модели прототипа, что в свою очередь позволит избежать программисту многих ошибок уже на этапе описания прототипа. Более того, явное задание графа переходов позволит значительно облегчить выполнение проверки его. Примечательно, что многие из рассмотренных выше языков соответствовали данному требованию.

Переиспользование компонентов кода является очень важным требованиям к автоматному языку. Оно позволяет решить проблему дублирования кода, а также является необходимым требованием при создании сложных систем. В тоже время, в отношении языка автоматного программирования это достаточно сильное требование, полноценная реализация которого может практически уничтожить все преимущества такого языка. Такую реализацию можно встретить в языке State Machine. Это привело к тому, что данный язык стал достаточно громоздким и для каждого состояния необходимо выполнить создание отдельного класса, объявление и инициализацию всех дополнительных переменных. Исходя из этого, является необходимым поиск компромисса между удобством проектирования и удобством кодирования.

Краткость и понятность синтаксиса языка является одним из важнейших требований. Практически все из рассмотренных ранее языков с явным заданием графа переходов имеют немного похожую структуру программы и частично синтаксис. В целом, от этого требования зависит эффективность использования языка автоматного программирования.

## 2.4 Верификация модели прототипа

В процессе верификации модели прототипа устанавливается факт работоспособности исходного прототипа архитектуры, а так же факт его соответствия предъявленным требованиям.

Переходя к модели прототипа архитектуры в виде конечного автомата можно провести как динамическую, так и статическую проверку корректности этого представления.

Динамическая проверка основывается на ведении протокола работы прототипа – запись всей последовательности состояний, в которых пребывал автомат, обработанных событии, значении входных и выходных переменных.

В статической проверке правильности прототипа можно условно выделить несколько уровней, в каждом из которых существуют специфические методы. Например, на синтаксическом уровне можно выделить такое правило: «Любой переход должен вести из одного состояния в другое». На уровне валидации можно выделить целый ряд поведенческих свойств, проверку которых будет несложно провести по графу переходов конечного автомата:

* *достижимость*: в любое состояние автомата есть путь из начального состояния, составленный из переходов. Иначе такое состояние является недостижимым для автомата, а, следовательно, не имеет смысла. Наличие недостижимого состояния в автомате безвредно, но может свидетельствовать об ошибке при построении;
* *непротиворечивость*: множество переходов из каждого состояния автомата должно быть непротиворечивым, то есть не должно существовать такого входного воздействия, при котором истинными становятся одновременно сразу несколько переходов из этого множества. Наличие такого рода противоречия во внутреннем представлении прототипа системы может привести к тому, что невозможно будет определить, какой из переходов срабатывает в какой ситуации;
* *полнота*: множество переходов из каждого состояния автомата должно быть полным. Иными словами, для каждого входного воздействия должен существовать хотя бы один переход из множества переходов данного состояния, условия которого будет истинным. Если данное условие не выполняется, то есть вероятность неправильной работы программы, поскольку во время выполнения может сложиться ситуация, когда автомат не сможет совершить переход.

Стоит отметить, что перечисленные выше свойства проверяются формально (без учёта семантики входных и выходных переменных). Дополнительное преимущество при проверке – это наличие спецификации.

В процессе верификации можно выделить несколько стадии [38]:

* моделирование;
* спецификация;
* верификация модели.

Каждая стадия включает в себя выполнение определённых действий, а так же наличие определённых трудностей. На стадии моделирования с одной стороны очень важно абстрагироваться от неважных деталей, но в тоже время не потерять значимые детали. Самой большой трудностью стадии спецификации будет сформулировать исчерпывающие и всеохватывающие требования к системе. На заключительной стадии верификации модели необходимо производить качественный анализ контрпримеров. Если алгоритм верификации не справляется со своей задачей, то есть вероятность что предложенная модель слишком объёмна, и её нужно упростить.

## 2.5 Model Checking

Среди методов статической проверки корректности можно выделить одно из направлений формальной верификации, которое в настоящий момент является достаточно популярным. Это верификация модели программы методом Model Checking [39, 40]. Этот подход к верификации представляет собой метод автоматической формальной верификации систем с конечным числом состояний. Данный подход позволяет установить удовлетворяет ли заданная модель системы формальным спецификациям.

Как правило, для верификации этим методом необходимо построение модели системы с выделением конечного числа состояний, а так же формальное описание требований к системе с применением темпоральной логики.

В общем случае, для того чтобы провести верификацию каркаса системы указанным методом, требуется:

1. построить формальную модель, приемлемую для инструментальных средств верификации моделей;
2. сформулировать требования к модели;
3. верифицировать модель. Если модель не соответствует указанным требованиям, то у пользователя есть возможность посмотреть путь с ошибкой – контрпример, который может помочь исправлению ошибки в модели. Ошибки могут происходить в результате некорректного задания самой модели или неправильных спецификаций. Контрпример помогает устранить ошибки, как в моделировании, так и в спецификации.

Основная трудность, характерная для проверки модели методом Model Checking, связана с эффектом комбинаторного взрыва в пространстве состояний. Такая проблема возникает в системах, состоящих из множества компонентов, которые взаимодействуют друг с другом, и в системах, которые обладают структурами данных, способными принимать большое число значений.

## 2.6 Верификатор SPIN

В качестве верификатора модели автором было решено использовать наиболее популярный на данный момент и проверенный временем верификатор SPIN [41-43].

Верификатор SPIN ("Simple Promela Interpreter") представляет собой инструмент для проверки корректности распределённых программных моделей в автоматном режиме. В основе верификатора SPIN лежит метод Model Checking. На сегодняшний день данный верификатор считается наиболее популярным, что позволяет надеяться на отсутствие в нем ошибок.

На вход SPIN поступает описание модели системы на языке Promela (Process Meta Language) [44], который поддерживает моделирование распределённых асинхронных алгоритмов в виде недетерминированного автомата. Требования к модели описываются посредством формул на специализированном языке темпоральной логики LTL (Linear Temporal Logic) [45]. Верификатор по модели системы строит модель Крипке [46], а по инверсии каждого требования – автомат Бюхи [47]. После этого строится пересечение модели Крипке и автомата Бюхи «на лету», то есть, не дожидаясь окончания полного построения модели Крипке, и, если пересечение не пусто, выдаётся трасса ошибки.

Кроме проверки модели SPIN также может быть использован в качестве симулятора, осуществляющего запуск работы модели системы по одному из возможных путей работы модели и представляя результат работы пользователю.

### 2.6.1 Язык Promela

Promela (Process or Protocol Meta Language) – это специализированный язык высокого уровня, который используется верификатором SPIN. Синтаксис Promela напоминает синтаксис языка С.

Модель на языке Promela состоит из следующих элементов:

* объявления типов данных;
* объявления каналов передачи переменных;
* объявления переменных;
* объявления процессов;
* процесса init.

Процесс отдалённо можно рассматривать как процедуру, выполняемую в отдельном потоке. Пример определения процесса:

proctype A()

{

byte state;

state = 3;

}

Ключевое слово proctype только объявляет процесс, но не запускает его. Первоначально в модели на языке Promela только один процесс будет запущен - процесс init, который должен быть всегда явно описан.

Тело процесса состоит из последовательности операторов. Операторы могут быть выполнимыми или заблокированными. Выполнимый оператор может быть выполнен немедленно, а заблокированный не может быть выполнен до тех пор, пока не станет выполнимым. Например, оператор y>5 может быть выполнен только в том случае, если y больше пяти. В противном случае такой оператор останавливает выполнение процесса до тех пор, пока значение y не станет больше 7. Некоторые операторы, например, оператор присваивания, выполнимы всегда.

Новые процессы могут быть запущены с использованием команды run, которая в качестве аргумента принимает название proctype, из которой в дальнейшем этот процесс будет создан. Оператор run можно использовать в объявлениях proctype и при объявлении процесса init. Благодаря этому есть возможность динамического создания процессов. Запущенный процесс заканчивает свою работу, как только достигнут конец тела proctype и все процессы, запущенные этим процессом, закончили свою работу.

Процессы могут иметь параметры и локальные переменные. Процесс может быть запущен в нескольких экземплярах, если у него стоит модификатор active.

Язык Promela содержит следующие типы данных:

* bit;
* bool;
* byte;
* short;
* int.

Кроме того, язык содержит операторы ветвления и цикла, синтаксис которых основан на охраняемых командах Дейкстры [48].

### 2.6.2 Модель Крипке

В методах верификации систем в качестве моделей обычно используют так называемую модель Крипке. Эта модель представляет собой недетерминированный конечный автомат и формально её можно описать над множеством атомарных высказываний AP (булевых выражений над множеством переменных, констант и предикатных символов) с помощью кортежа вида:

*M = (S, S0, R, L)*, в котором

S — конечное множество состояний,

S0 ⊆ S — множество начальных состояний,

R ⊆ S X S — отношение перехода,

L: S → 2AP — функция пометок. Функция пометок L для каждого состояния s ∈ S определяет множество L(s) всех атомарных утверждений верных в s.

Модель Крипке представляется в виде графа переходов (или диаграммы переходов) – ориентированного графа, вершины которого описывают достижимые состояния, а ребра - переходы из состояния в состояние. Функция пометок сопоставляет каждой вершине множество свойств, которые выполняются в соответствующем состоянии.

Описанная модель приспособлена для верификации. Для записи требований к ней используются языки темпоральной логики: LTL, CTL, CTL\* и другие.

Построим пример модели Крипке (см. Рисунок 2.2). Пусть множество атомарных высказываний AP={q,p}. Переменные q и p могут моделировать какие-либо произвольные логические свойства для данной модели. Исходя из определения модели Крипке, описание модели можно провести с помощью кортежа *M = (S, S0, R, L)*, в которой

* S = {s1, s2, s3},
* S0={ s1},
* R = {(s1,s2),(s2,s1)(s2,s3),(s3,s3)},
* L = {(s1, {q, p}), (s2, {p}), (s3, {q})}.

В построенной модели M можно выделить сценарий ρ = {s1,s2,s1,s2,s3,s3,s3,...} и соответствующий этому пути сценарий w = {{q,p},{p},{q,p},{p},{q},{q},{q},...}. Сценарий w является -*языком* над алфавитом 2AP. Таким образом, все сценарии, которые может выполнять модель M, принадлежат языку ({q,p}{p})\*({q})ω∪({q,p}{p})ω.



Рисунок 2.2 – Пример модели Крипке

Исходя из определения модели Крипке и описанного выше абстрактного понятия модели прототипа архитектуры, можно сделать вывод об их взаимно-однозначном соответствии. Действительно, обе представленные модели содержат в своём описании множество состояний,

### 2.6.3 Язык темпоральной логики LTL

Темпоральная логика LTL(Linear Temporal Logic или Linear-time Temporal Logic) представляет собой вид модальной темпоральной логики, которая кроме стандартных логических связок, переменных и предикатов содержит модальные операторы, ссылающиеся на время.

Язык LTL состоит из множества атомарных высказываний p1, p2, … AP, логических связей, /\, \/, → и темпоральных операторов.

Пусть *φ* — правильно построенная формула. Тогда

* X*φ* (в следующем состоянии *φ* верно — neXt);
* G*φ* (*φ* верно всегда — Globally);
* F*φ* (*φ* когда-нибудь будет верно — Finally);
* *ψ*U*φ* (*ψ* будет верно до тех пор, пока не станет верно *φ* — Until);
* *ψ*R*φ* (*φ* будет верно до момента, когда *ψ* станет верно и включая его — Release. Если *ψ* никогда не станет верным, то значит *φ* всегда будет верным).

тоже правильно построенные формулы.

Операторы G и F необходимы для упрощения формул. Их можно выразить через оператор U:

* F*φ* 1U*φ*;
* G*φ* F*φ*.

Формулы *LTL* интерпретируются через исполнение системы переходов в модели Крипке. Если все пути из начального состояния удовлетворяют формуле *φ*, то будем говорить, что поведение системы удовлетворяет формуле *φ*.

### 2.6.4 Автомат Бюхи

Пусть AP – это множество атомарных высказываний. Автоматом Бюхи над алфавитом 2AP называется кортеж вида:

*A = (Q, q0, δ, F)*, в котором

*Q* — конечное множество состояний,

*q0* — начальное состояние,

*δ⊆ Q X 2AP XQ* — функция переходов,

*F⊆ Q* — множество допустимых состоянии.

Доказано, что для любой формулы на языке темпоральной логики LTL можно построить автомат Бюхи, который её выполняет [49, 50]. Более того, построение этого автомата может проходить автоматически.

Рассмотрим пример. Пусть есть LTL-формула [] (p U q). Эта формула обозначает, что всегда гарантировано, что условие *p* остаётся истинным, по крайней мере, до тех пор, пока не станет верным условие *q*.

Верификатор SPIN транслирует эту формулу в конструкцию never claim:

never {/\* [] (p U q) \*/

T0\_init:

if

:: ((q)) -> goto accept\_S9

:: ((p)) -> goto T0\_init

fi;

accept\_S9:

if

:: ((((p)) || ((q)))) -> goto T0\_init

fi;

}

Представленная конструкция соответствует автомату Бюхи, представленному на Рисунке 2.3. С помощью двойной линии обозначено допустимое состояние.

С помощью автомата Бюхи можно верифицировать модель Крипке. С точки зрения верификации автоматных моделей – это наиболее удобный способ, позволяющий при верификации и спецификации почти полностью ограничиться понятием конечный автомат.



Рисунок 2.3 – Автомат Бюхи для LTL-формулы *G(p U q)*

## 2.7 Выводы

Поскольку целью работы является разработка системы, позволяющей производить проектирование и тестирование каркасов систем, необходимо определить средства, используемые в качестве следующих компонентов:

* Верификатор;
* Целевой язык автоматного программирования;
* Язык описания моделей прототипов.

В качестве верификатора для проверки моделей был выбран SPIN. Верификатор на основе полученной модели будет составлять программу на языке программирования C, соответствующую предъявленной модели и требованиям, которая далее будет компилироваться с помощью компилятора gcc [51]. В случае обнаружения ошибки в исследуемом прототипе будет составляться контрпример, отражающий её путь возникновения. Полученный контрпример необходимо привести к виду, понятному конечному пользователю, то есть привести к тому же языку, на котором описан прототип системы.

Для осуществления проверки прототипов систем в качестве целевого автоматного языка был выбран язык Promela. Выбор указанного языка обусловлен использованием его верификатором SPIN для проверки моделей прототипов. Данный язык обладает достаточно широким набором возможностей и позволяет описывать даже распределенные и параллельные системы. Такие большие возможности языка привели к тому, что её использование в качестве языка описания моделей требует наличия определённых знании и опыта у разработчика. Таким образом, Promela в связи с высоким порогом входа не может быть использован в качестве языка описания каркасов программных систем.

Для удобства описания необходимо использовать простой и удобный язык, который позволял бы задавать каркас системы с учетом объектно-ориентированного подхода. В силу выявленных недостатков у существующих языков автоматного программирования было принято решение разработать язык описания моделей прототипов, который будет удовлетворять предъявленным требованиям. Синтаксис предложенного автором данной работы языка приведен в разделе 3.

# 3 Реализация системы

## 3.1 Общее описание

В данной работе автором предлагается система проектирования и тестирования каркасов программных продуктов (см. Рисунок 3.1), которая представляет собой программный комплекс, предоставляющий возможность производить верификацию каркасов с использованием верификатора SPIN.



Рисунок 3.1 – Архитектура системы

Система состоит из трёх основных подсистем:

* подсистемы лексического и синтаксического анализа, позволяющей на основе грамматического разбора входного прототипа получить соответствующее ему абстрактное синтаксическое дерево;
* подсистемы построения графа потока управления, реализующей сбор информации из абстрактного синтаксического дерева и построение графа вызовов модели прототипа;
* подсистемы кодогенерации, отвечающей за генерацию кода прототипа на целевом языке.

## 3.2 Подсистема лексического и синтаксического анализа

### 3.2.1 Структура подсистемы лексического и синтаксического анализа

Подсистема лексического и синтаксического анализа (см. Рисунок 3.2), разработанная автором работы, представляет собой программный комплекс, обеспечивающий выполнение лексического и синтаксического разбора для всех входных файлов.



Рисунок 3.2 – Структура подсистемы

### 3.2.2 Описание языка Proto

Поскольку в результате рассмотрения существующих автоматных языков описания не нашлось языка, удовлетворяющего предъявленным требованиям, автором было решено разработать новый язык описания для моделей прототипов программных продуктов. Главная задача такого языка состоит в том, чтобы упростить описание таких моделей прототипов. Кроме того, наличие простого и понятного средства для описания каркасов программных систем позволит снизить порог входа в область верификации, то есть сделать процесс верификации моделей более доступным для конечного пользователя.

Возможности разработанного языка позволяют с использованием синтаксиса, близкого к синтаксису языка программирования Java [52], описать модель исходного прототипа, а именно все события, состояния, переходы и другие элементы модели. Рассмотрим полное описание возможностей языка Proto.

#### 3.2.2.1 Синтаксис языка

Опишем синтаксис языка Proto, его ключевые слова и конструкции. В целом, в основу данного языка положен язык программирования Java. В качестве примера схожести можно привести метод объявления классов и интерфейсов, выделение вложенных конструкции фигурными скобками и прочее.

Синтаксис языка содержит следующие ключевые слова:

* interface – используется для объявления интерфейсов классов;
* class – ключевое слово, используемое для создания классов. Поддержка реализации интерфейсов реализована с помощью оператора ‘<’, наследования (в том числе и множественного) с помощью оператора ‘<<’;
* prototype – ключевое слово для выделения начала секции описания требований к модели прототипа;
* операторы, необходимые для описания формул темпоральной логики LTL:
  + finally;
  + globally;
  + until;
  + release;
* основные типы данных:
  + void;
  + state – особый типа данных, объявляющий состояния в модели прототипа. Каждое состояние создаёт новый уровень вложенности и обязательно содержит внутри своё описание. Состояние может содержать объявление вложенных автоматов, вложенных состояний, а также переходов. Для удобства в качестве начального состояния всегда будем использовать состояние main, а в качестве конечного – finalize. Начальное состояние обязательно содержит в своём описании переход в следующее состояние, а конечное состояние соответственно не должно содержать никаких переходов;
  + number – обобщённый числовой тип данных;
  + bool;
  + string;
  + object – базовый тип для всех типов языка;
* массивы;
* встроенные операторы:
  + print – простой вывод данных;
  + die – обеспечивает аварийное прерывание модели прототипа и вывод сообщения;
  + return;
  + random – встроенный оператор генерации случайного числа в заданном диапазоне;
* операторы циклов и условий: if, for, do, while.

В языке proto реализована поддержка использования комментариев подобно языку программирования java, то есть допустимы два вида комментариев – однострочные и многострочные.

#### 3.2.2.2 Грамматика языка

Язык описания моделей Proto может быть задан с помощью следующей грамматики:

program : (statement)\* ;

statement : interface\_decl|class\_decl|proto\_decl ;

proto\_decl: 'prototype' ID '('parameters')' '{'(spec\_decl)\*'}’;

spec\_decl: UNARY\_SPEC+ '{'spec\_expression'}' ';' |

'{'spec\_expression'}' BINARY\_SPEC '{'spec\_expression'}' ';';

UNARY\_SPEC: 'finally'|’globally’;

BINARY\_SPEC: 'until'|'release'|’if’;

spec\_expression: or\_spec;

or\_spec: and\_spec ('or' and\_spec)? ;

and\_spec: not\_spec ('and' not\_spec)? ;

not\_spec: ('not')? spec;

spec: one\_spec ('=='|'! =' one\_spec)\*;

one\_spec: ID | '(' spec\_expression ')’;

interface\_decl: 'interface' ID '{'(method\_decl)\*'}’;

method\_decl: type ID '('arguments')' ';’;

arguments: (type ID (',' type ID)\*)? ;

parameters: (big\_expression (',' big\_expression)\*)? ;

type: 'void'|'state' array|'number' array|'bool' array|

'string' array|'object' array |ID array;

array: ('[' ']')\*;

class\_decl:'class' ID '('arguments')' ('<' ID (','ID)\*)?

('<<' ID)? '{'(body)\* '}’;

body: method|field;

method: type ID ‘('arguments')' '{' (operator)\* '}’;

field: type ID ('=' big\_expression)? ';' ;

operator: field|assignment ';'|buildin\_operator|call ';'|

if\_operator|for\_operator|while\_operator|do\_operator

|'{' (operator)\* '}’;

buildin\_operator: print | die | return\_operator;

die: 'die' big\_expression ';’;

print: 'print' big\_expression ';’;

return\_operator: 'return' big\_expression ';’;

assignment: ID ('['big\_expression']')\* '=' big\_expression;

big\_expression: or\_expression;

or\_expression: and\_expression ('or' and\_expression)? ;

and\_expression: not\_expression ('and' not\_expression)? ;

not\_expression: ('not')? expression;

expression: relation (('=='|'! =') relation)? ;

relation: summand (('>'|'<'|'<='|'>=') summand)? ;

summand: multiplier ('+' multiplier |'-' multiplier)\*;

multiplier: simple\_expression

('\*' simple\_expression|'/' simple\_expression)\* ;

simple\_expression: ID ('['big\_expression']')\*|call|INT|

STRING|'['parameters']'|'('big\_expression')'|'nan'|'nil'|'new' ID '('parameters')'|'random' (ID| INT);

call: ID '('parameters')’;

if\_operator: 'if' '('big\_expression')' operator ('else' operator)? ;

for\_operator: 'for' '('assignment';'big\_expression';'assignment')'

operator;

while\_operator: 'while' '('big\_expression')' operator;

do\_operator: 'do' '{'operator'}' 'while' '('big\_expression')' ';’;

#### 3.2.2.2 Пример программы на языке proto

В качестве иллюстрации использования языка рассмотрим описание программы, представляющей из себя имитационную модель некоторого объекта. Поскольку для любой программной системы может быть построен соответствующий ей конечный автомат, то поведение любой программы может быть описано в терминах перехода конечного автомата из состояния в состояние, а, следовательно, эта программа может быть описана на предлагаемом языке Proto.

В качестве моделируемой программы возьмём прототип, реализующий имитационную модель автоматической коробки передач [53] (см. Рисунок 3.3). Выбор этого примера объясняется как простотой реализации данной имитационной модели, так и наличием достаточно большого количества состоянии, что позволяет продемонстрировать основные возможности предлагаемого языка.



Рисунок 3.3 – Модель имитации работы автоматической коробки передач

Прототип должен удовлетворять следующим требованиям:

1. управление коробкой передач автомобиля может быть выполнено посредством трёх основных действий:
   1. включение зажигания;
   2. переключение направления движения;
   3. удержание педали газа.
2. двигатель может быть запущен при помощи нажатия кнопки ВКЛ (ON);
3. пользователь может выбирать направление движения:
   1. вперёд;
   2. назад;
   3. нейтраль – двигатель работает, но автомобиль не двигается.
4. при удержании газа двигатель начинает наращивать обороты. При достижении отметки 2000 оборотов происходит переключение на более высокую передачу;
5. пользователь может заглушить двигатель, только находясь на нейтральной передаче.

Рассмотрим описание на языке Proto, соответствующее построенной модели (см. Рисунок 3.4).

Синтаксис языка позволяет производить описание и реализацию интерфейсов, в данном случае объявлен интерфейс Driver и объявлена его реализация DefaultDriver, описывающая поведение водителя. Далее объявляется главный класс AutoGear, который обязательно должен наследовать базовый класс Prototype. В главном классе приводится описание основных состояний модели, а также устанавливаются отношения между ними.



Рисунок 3.4 – Код примера на языке Proto

Для описываемой модели системы можно выделить следующие состояния:

* main – является начальным состоянием системы по умолчанию. Из данного состояния осуществляются переходы в другие состояния, в нашем случае в состояние *sleep*;
* sleep – состояние, в которое прототип переходит после начального. В данном состоянии происходит ожидание включения двигателя;
* neutral – состояние, в котором двигатель работает, но автомобиль не двигается, поскольку ожидает выбора водителем направления движения, в зависимости от которого возможно начало движения вперёд (состояние gear1) или назад (состояние reverse);
* reverse – состояние, характеризующееся движением автомобиля назад. Движение автомобиля назад можно остановить с помощью переключения на нейтральную передачу (переход в состояние neutral);
* gear1 – состояние, для которого характерно нахождение коробки передач на первой передаче, а также начало движения автомобиля вперёд. При наборе достаточного количества оборотов происходит переключение на повышенную передачу (переход в состояние gear2);
* gear2 - gear5 – состояния, при которых коробка передач находится во второй-пятой передаче соответственно. Как и для первой передачи при достижении определённого количества оборотов происходит переключение;
* finalize – является завершающим состоянием любой модели прототипа. Для данной ситуации переход в заключительное состояние связан с событием «Выключение двигателя», которое может произойти только во время нахождения в состоянии neutral, то есть после включения двигателя.

### 3.2.3 Выбор средств реализации

В процессе выбора инструментов реализации подсистемы лексического и синтаксического анализа автором были рассмотрены популярные на сегодняшний день технологии построения и разбора предметно-ориентированных языков. Автором были рассмотрены следующие решения:

* JavaCC [54] — средство создания классов на языке Java для проверки и разбора структурированного текста. JavaCC позволяет нам задавать грамматики в виде, сходном с EBNF (Extended Backus–Naur Form – Расширенная форма Бэкуса-Наура), делая лёгким перенос грамматик, записанных в такой форме, в формат JavaCC. Этот инструмент является самым популярным инструментом для создания парсеров на Java с большим набором готовых грамматик;
* GNU bison [55] — программа, предназначенная для автоматического создания синтаксических анализаторов по данному описанию грамматики. Данная программа во многом совместима с подобной программой yacc. Обычно используется в комплексе с лексическим анализатором flex [56]. Bison используется для описания грамматики, построенной на базе алфавита токенов, и используется для генерации программы (кода на языке C, C++ или Java), которая получает на вход поток токенов и находит в этом потоке структурные элементы (нетерминальные токены) согласно заданной грамматике;
* Yacc [57] — стандартный генератор синтаксических анализаторов (парсеров) в Unix-системах. Он генерирует парсер на основе аналитической грамматики, описанной в нотации BNF (форма Бэкуса-Наура) или контекстно-свободной грамматики. На выходе yacc выдаётся код парсера на языке программирования Си;
* Сосо/R [58] — программа генерации компиляторов или интерпретаторов языка. Программа читает файл с атрибутивной грамматикой исходного языка в форме EBNF и генерирует ряд файлов для этого языка: лексический анализатор (сканер), синтаксический анализатор (парсер), информационные файлы (листинг, таблица лексем языка). Сканер работает как конечный автомат. Парсер использует методику нисходящего рекурсивного спуска [59];
* ANTLR [60] — генератор парсеров, позволяющий автоматически создавать программу-парсер и лексический анализатор на одном из целевых языков программирования (C++, Java, C#, Python, Ruby) по описанию LL(\*)-грамматики на языке, близком к РБНФ (Расширенная форма Бэкуса-Наура).

В итоге, был выбран генератор парсеров ANTLR в силу доминирующего превосходства над аналогами. ANTLR, как и многие другие подобные средства, состоит из библиотеки классов, облегчающих основные операции при разборе (буферизация, поиск и т. д.), и утилиты, генерирующей код парсера на основе файла, описывающего грамматику разбираемого языка. Сам ANTLR написан на Java, но позволяет генерировать код на Java и C++. В качестве источника данных, используется класс потока, что позволяет разбирать данные, находящиеся в файле, в памяти процесса или приходящие по сети. Результатом работы ANTLR являются два класса — лексер и парсер. Лексер разбивает поток символов на поток токенов в соответствии с правилами, а парсер обрабатывает поток токенов в соответствии с другими правилами.

Кроме того, во время работы над системой для удобства использования инструментом ANTLR в рамках проекта всей системы был использован фреймворк для автоматизации сборки Apache Maven [61].

### 3.2.4 Расширенная форма Бэкуса-Наура

Расширенная форма Бэкуса — Наура [62] (англ. Extended Backus–Naur Form (EBNF)) представляет собой формальную систему описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. Данная форма используется для описания контекстно-свободных формальных грамматик. Является расширенной переработкой формы Бэкуса-Наура и отличается от неё использованием улучшенных конструкций, позволяющих при той же выразительной способности упростить и сократить в объёме описание грамматики. Данная форма описания грамматики принята в качестве стандарта Международной организацией по стандартизации (ISO).

### 3.2.5 Структура абстрактного синтаксического дерева

Подсистема лексического и синтаксического анализа в качестве выходных данных формирует абстрактное синтаксическое дерево [63], структура которого была разработана в процессе дипломного проектирования. Абстрактное синтаксическое дерево в данном случае будет включать в себя только те конструкции модели прототипа, которые важны для дальнейшего анализа и генерации кода (см. Рисунок 3.5). Среди таких конструкции можно выделить:

* Секцию описания спецификации требований prototype,
* Описание спецификации с ключевыми словами finally, globally, until и другими,
* Декларация классов и методов с учетом области видимости,
* Описание переменных,
* Вызовы функции и состояний.



Рисунок 3.5 - Пример абстрактного синтаксического дерева, соответствующего модели прототипа

Система ANTLR позволяет в качестве выходного значения получить абстрактное синтаксическое дерево, которое будет соответствовать входному файлу для заданной грамматики. Формирование такого дерева специфицируется в исходной грамматике при помощи специальных операторов и *rewrite*-правил [64].

Операторы позволяют сделать явное указание на то, что данный узел должен быть включён в дерево или исключён из него. Оператор «!» указывает на то, что данный узел или поддерево (в случае целого правила) не должны включаться в дерево. А оператор «^» делает следующий за ним узел корнем поддерева

Синтаксис rewrite-правил более мощный по сравнению с операторами. Использования таких правил в общем случае достаточно для построения структуры дерева. В то время как грамматика указывает парсеру на то, как должен быть распознан входной файл, такие правила отвечают за генерацию выходных данных. Таким образом, ANTLR определяет, как сопоставить входные и выходные данные грамматике.

## 3.3 Подсистема построения графа потока исполнения

### 3.3.1 Общее описание

Подсистема построения графа потока исполнения реализует механизмы разбора абстрактного синтаксического дерева, полученного на этапе работы подсистемы лексического и синтаксического анализа, и непосредственно построения необходимого графа. Под графом потока исполнения [65] (от англ. control flow graph, CFG) будем понимать множество всех возможных путей исполнения программы, представленное в виде графа.

### 3.3.2 Алгоритм построения графа потока исполнения по абстрактному синтаксическому дереву программы

В основе алгоритма лежит обход абстрактного синтаксического дерева, построенного подсистемой лексического и синтаксического анализа. Для упрощения построения графа в исходную грамматику включены специальные вставки, устанавливающие взаимно однозначное соответствие между узлами и их типами. Перечислим типы узлов, которые необходимы на этапе построения графа потока исполнения:

* вызов функции или состояния – тип узла CallNode;
* объявление класса – тип узла ClassNode;
* декларация нового метода – тип узла MethodNode;
* декларация секции описания требований к модели прототипа – сопоставляется типу PrototypeNode;
* описание одного требования к модели – тип узла SpecNode;
* объявление нового состояния модели прототипа – тип узла StateNode.

Вставки в грамматику языка, устанавливающие тип узла в синтаксическом дереве выглядят следующим образом:

class\_decl: 'class' ID '(' arguments ')' (‘<' ID ( ',' ID)\*)?

(‘<<' ID)? '{' (body)\* '}' -> ^ (ID<ClassNode> (body)\*);

В данном примере с помощью вставки указывается, что при разборе этого правила грамматике должно быть создано новое поддерево с корневым узлом, соответствующим типу ClassNode, то есть объявлению класса.

Алгоритм построения графа потока исполнения может быть разделён на два основных этапа:

1. поиск всех состоянии – осуществляется при помощи обхода дерева и поиска всех узлов, соответствующий типу StateNode;
2. формирование путей между узлами графа – основывается на обходе дерева
   1. если текущий узел – это вызов функции или состояния, и он является поддеревом узла состояния
      1. если текущий узел является состоянием – добавляем путь от корня поддерева, содержащего текущий узел к текущему узлу;
      2. если текущий узел представляет собой вызов метода – проверяем, нет ли внутри этого метода вызова других состоянии, и строим соответствующие пути;
   2. если текущий узел – это декларация секции описания требований, то обходим соответствующее ему поддерево и формируем список спецификации.

Результатом работы алгоритма является граф потока исполнения, отражающий последовательность вызовов (см. Рисунок 3.6).



Рисунок 3.6 – Пример графа потока исполнения

## 3.4 Подсистема кодогенерации

Разработанная автором подсистема кодогенерации реализует заключительную часть в работе системы, заключающуюся в генерации кода на целевом языке (см. Рисунок 3.7). На основе графа исполнения, построенного описанным ранее способом, строится представление на целевом языке. Как уже было сказано ранее, в качестве целевого языка на данном этапе будем использовать автоматный язык Promela.

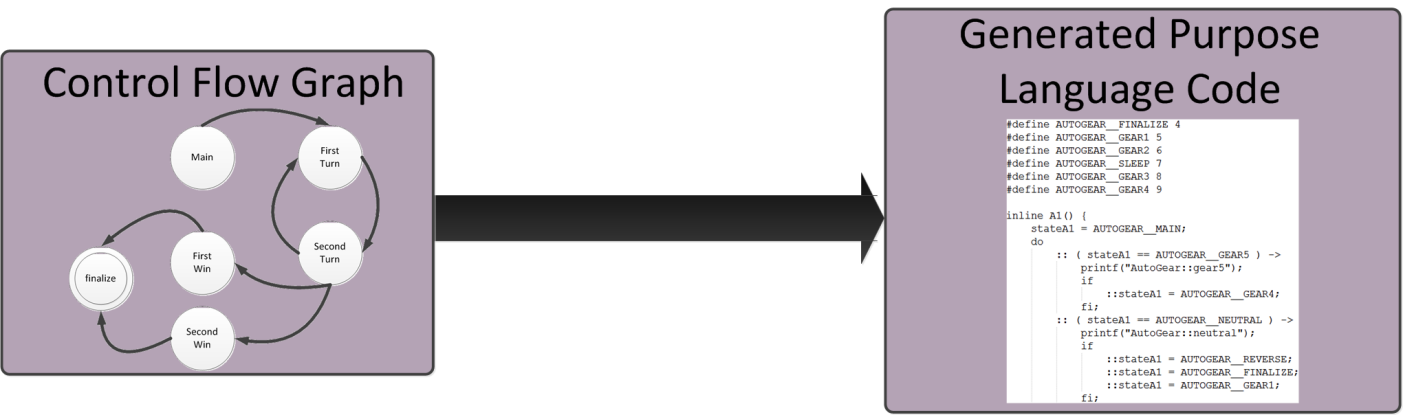


Рисунок 3.7 – Подсистема кодогенерации

### 3.4.1 Построение модели на языке Promela

Для представления каркаса программного продукта на целевом языке автором был разработан и использован специализированный алгоритм построения соответствующей модели на основе графа потока управления исходного прототипа [66]. В работе алгоритма условно можно выделить два основных этапа – это подготовка и построение.

Подготовка к построению является начальным этапом в построении. На данном этапе для каждого автомата Ai вводится переменная stateAi. На языке Promela это выглядит следующим образом:

int stateAi;

Каждому состоянию должен быть присвоен уникальный номер, используя сквозную нумерацию для всех автоматов. Переход автомата Ai в новое состояние с номером k будет осуществляться присваиванием соответствующей автомату переменной Ai числа k:

stateAi = k;

Событие на переходе между состояниями exx (где xx – номер события) может быть описано в модели следующим образом:

lastEvent = xx;

Для всех автоматов системы создадим специальную функцию-макрос:

inline Ai(){

/\* тело функции \*/

}

Начальный шаг на этапе построения модели заключается установке взаимно-однозначного соответствия между состояниями автомата Ai и их порядковыми номерами. Затем выполняется выделение начального состояния. В качестве начального состояния моделей на разработанном языке Proto используется специализированное состояние MAIN, которое соответственно должно встречаться в описании каждого каркаса. Номер, соответствующий состояния MAIN, присваивается переменной stateAi.

Главный цикл программы строится следующим образом (s1,..,sk – номера состояний автомата Ai):

do

:: (stateAi == s1) ->

:: (stateAi == s2) ->

…

:: (stateAi == sk) ->

od;

При наличии состояний с вложенными автоматами, для каждого такого состояния sj в условие *(stateAi == sj)* дописывается вызов соответствующей функции этого автомата:

Am();

Для каждого состояния sj нужно найти все возможные состояния, в которые возможны переходы (sj, sl). Если из состояния есть хотя бы один переход, то к условию (stateAi == sj) дописывается конструкция if. Для каждого перехода (sj, sl) в конструкцию if добавляется следующее:

::stateAi = sl;

Если переходу (sj, sl) кроме того соответствует некоторое событие exx, то после предыдущего выражения необходимо дописать следующее выражение:

lastEvent = exx;

Если состояние si – конечное, то в условие (stateAi == si) необходимо добавить инструкцию завершения цикла:

break;

### 3.4.2 Преобразование LTL-формул

С помощью языка Proto можно описать требования к поведению системы значительно проще по сравнению с описанием непосредственно на языке темпоральной логики LTL. При реализации описания требований в разработанном автором языке важным аспектом являлась поддержка лёгкой трансляции в синтаксис LTL.

В качестве примера рассмотрим требования к модели прототипа имитации автоматической коробки передач:

prototype AutoGear () {

finally {AutoGear::finalize};

globally finally {AutoGear::neutral};

{AutoGear::neutral} until {AutoGear::finalize};

}

Приведённым выше требованиям соответствуют следующие правила на языке LTL c поддержкой нотации синтаксиса верификатора SPIN:

<> {stateA1 == AutoGear::finalize};

[] <> {stateA1 == AutoGear::neutral};

{stateA1 == AutoGear::neutral} U {stateA1 == AutoGear::finalize};

Чтобы преобразовать требования к необходимому виду нужно выполнить следующие преобразования:

1. все элементарные высказывания должны быть записаны в фигурных скобках;
2. ключевые слова finally, globally, until и release преобразуются к нотации SPIN, а именно в операторы <>, [], U, R соответственно;
3. должен поддерживаться синтаксис целевого языка;
4. в случае неправильной формулы выводится сообщение «ERROR: Wrong specification description!».

Для того чтобы получить вид формул, необходимый для тестирования с помощью верификатора SPIN, нужно отрицание построенных формул подать на вход верификатору с ключом –f. Этот ключ запускает верификатор в режиме генерации конструкции never claim – представления автомата Бюхи в виде специального процесса на Promela. Never claim используется для описания свойства, которое никогда не должно быть выполнено во время верификации модели.

Пример конструкции never claims, сгенерированной по команде spin -f "!(<>{stateA1 == AUTOGEAR\_\_FINALIZE})":

never { /\*! (<> {stateA1 == AUTOGEAR\_\_FINALIZE}) \*/

accept\_init:

T0\_init:

if

:: (! ((stateA1 == AUTOGEAR\_\_FINALIZE))) -> goto T0\_init

fi;

}

### 3.4.3 Типовые требования к автоматным моделям

Существует набор требований, который должен выполняться для любой корректной модели системы. Эти требований могут быть полезны для проведения базовой верификации при отсутствии специфических для конкретной модели требований. Сформулируем некоторые из типовых требований [67]:

* завершение работы – главный автомат когда-нибудь окажется в конечном состоянии. В рамках автоматной модели это требование можно записать на языке темпоральной логики как

F (state == end\_state),

где state – текущее состояние главного автомата, end\_state – конечное состояние;

* постоянство – некоторое свойство *p* с какого-то момента будет выполнено всегда. Данное требование может быть применимо к любой модели. Вид требования на языке темпоральной логики:

FG (*p*);

* прогресс – в любой момент времени выполнения автомат может прийти в состояние *s* или произойдёт событие *exx*, что на языке LTL выглядит следующим образом:

GF (state == *s*) или GF (lastEvent == *exx)*;

* корректное завершение работы – зависит от конкретной модели.

## 3.5 Plug-in для Eclipse

Для удобства описания моделей прототипа с использованием языка Proto автором был создан специализированный полнофункциональный текстовый редактор в виде надстройки над существующей средой разработки Eclipse. Редактор предоставляет возможность для эффективной разработки моделей, а именно обладает следующими свойствами: автоматическое завершение ввода и исправление ошибок, валидация автоматной программы в процессе её создания и отладка программы. Многие современные среды разработки программного обеспечения имеют подобные возможности, так как их наличие позволяет значительно ускорить и упростить процесс создания программ.

Для этой цели можно воспользоваться инструментом Xtext [68], который представляет собой средство для разработки языков программирования и предметно-ориентированных языков. В основе реализации редактора лежит описание грамматики разработанного автором языка Proto в расширенной форме Бэкуса-Наура. Таким образом, требуемый текстовый редактор реализован в виде встраиваемого модуля для интегрированной среды разработки Eclipse (см. Рисунок 3.8).



Рисунок 3.8 – Plug-in для Eclipse

# Заключение

В результате работы над дипломным проектом была выдвинута и исследована идея построения системы проектирования и тестирования каркасов программных продуктов. Результаты работы опубликованы и доложены на IX Всероссийской научно-технической конференции «Наука и Молодежь 2012» [69]. Были изучены современные языки автоматного программирования, произведена оценка их эффективности и применимости согласно выдвинутой модели требований, сделаны выводы о неготовности существующих решений удовлетворять выдвигаемым к ним требованиям, и предложен новый язык для описания каркасов программных систем.

Была разработана и формализована модель системы проектирования и тестирования, легко расширяемая и полностью удовлетворяющая выдвинутой модели требований. На основании формализованной модели был спроектирован и реализован каркас системы проектирования и тестирования прототипов архитектур, с применением современных подходов и технологий программирования.

Спроектированное и разработанное в рамках дипломного проекта программное обеспечение каркаса системы проектирования может быть использовано для построения практически любого вида программных систем. В первую очередь проект ориентирован на использование в коммерческом секторе для обеспечения уменьшения затрат на разработку ПО за счёт минимизации ошибок на этапе проектирования архитектуры.

Можно выделить несколько путей развития проекта:

* поддержка верификация распределённых и параллельных систем;
* генерация других целевых языков для проверки другими верификаторами;
* совершенствование компонентов и оптимизация алгоритмов базовой платформы;
* сопровождение системы, информационная и техническая поддержка пользователей.

# Список использованных источников

1. Макконелл С. Совершенный код. Мастер-класс. Практическое руководство по разработке программного обеспечения / С. Макконелл. – Издательство «Русская редакция», 2011. – 896 с: ил.
2. Ананьев П.И. Технология разработки программного обеспечения: Учебное пособие/ Алт. госуд. технич. ун-т им. И.И. Ползунова.- Барнаул: 2009.-183.- ил.
3. Вигерс К. Разработка требований к программному обеспечению / К. И. Вигерс – Microsoft Press, Русская Редакция, 2004. – 576 с.
4. Прототипирование программного обеспечения [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Software\_Prototyping
5. Прототип [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Prototyping
6. Ларман К. Применение UML и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и итеративную разработку / К. Ларман – ООО «И. Д. Вильямс», 2009. – 616 с.
7. Len B. Software Architecture in Practice, Second Edition / B. Len, P. Clements, R. Kazman – Addison-Wesley, 2003. – 560 c.
8. Хант Э. Программист-прагматик. Путь от подмастерья к мастеру / Э. Хант, Д. Томас - СПб.: Питер, 2010. – 830 с.
9. Robert M. Agile Principles, Patterns, and Practices in C# / M. Robert, M. Micah – Prentice Hall, 2006. – 768 c.
10. Фаулер М. Рефакторинг. Улучшение существующего кода / М. Фаулер, К. Бек, Д. Брант, В. Апдайк, Д. Робертс – Символ-Плюс, 2008. – 432 с.
11. Larman C. Agile and Iterative Development: A Manager's Guide / C. Larman – Addison Wesley, 2003. – 368 с.
12. Manifesto for Agile Software Development [Электронный ресурс] – Режим доступа: http://agilemanifesto.org/
13. Beck K. Extreme Programming Explained: Embrace Change / K. Beck, C. Andres – Addison Wesley Professional, 2004. – 224 c.
14. Clinton K. Agile game development with Scrum / K. Clinton – Addison Wesley, 2010. – 329 c.
15. Extreme perl [Электронный ресурс] – Режим доступа: http://www.extremeperl.org/bk/home
16. Шалыто А. А. Автоматное программирование / Н. И. Поликарпова, А. А. Шалыто – СПб.: Питер, 2011.- 176 с: ил.
17. Формальная верификация [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Formal\_verification
18. Грис Д. Наука программирования / Д. Грис – М.:Мир, 1984. – 416 с: ил.
19. Непомнящий В. Прикладные методы верификации программ / В. Непомнящий, О. Рякин – М.: Радио и связь, 1988. – 250 c.
20. Кларк Э. Верификация моделей программ. Model Checking / Э. Кларк, О. Грамберг, Д. Пелед – М.:МЦНМО, 2002. – 416 с.
21. Pnueli A. The temporal logic of programs / A. Pnueli – Proceedings of the 18th IEEE Symposium on Foundation of Computer Science, 1977. – 46-57 c.
22. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч, Р. Максимчук, М. Энгл, Б. Янг, Д. Коналлен, К. Хьюстон – Вильямс, 2010. – 720 с: ил.
23. Темпоральная логика [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Temporal\_logic
24. Язык AsmL [Электронный ресурс] – Режим доступа: http://research.microsoft.com/fse/asml
25. Язык FSMGenerator [Электронный ресурс] – Режим доступа: http://fsmgenerator.sf.net
26. Язык SMC [Электронный ресурс] – Режим доступа: http://smc.sf.net
27. Шамгунов Н. Н. Разработка методов проектирования и реализации поведения программных систем на основе автоматного подхода. Диссертация на соискание ученой степени кандидата технических наук / Н. Н. Шамгунов – СПб.: СПбГУ ИТМО, 2004. – 195 с.
28. Цымбалюк Е. А. Текстовый язык автоматного программирования. Магистерская диссертация / Е. А. Цымбалюк – СПб.: СПбГУ ИТМО, 2008. – 72 с.
29. Степанов О. Г. Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby / О. Г. Степанов, А. А. Шалыто, Д. Г. Шопырин – Информационно-управляющие системы № 4, 2007. – 22–27 c.
30. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления / А. А. Шалыто – СПб.: Наука, 1998. – 617 с.
31. Шалыто А. А., Туккель Н. И. SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем [Электронный ресурс] – Режим доступа: http://is.ifmo.ru/download/switch.pdf
32. Язык Ruby [Электронный ресурс] – Режим доступа: http://ru.wikipedia.org/wiki/Ruby
33. Гуров В. С., Мазин М. А., Шалыто А. А. Текстовый язык автоматного программирования / Тезисы докладов международной научной конференции,посвященной памяти профессора А. М. Богомолова «Компьютерные науки и технологии». Саратов: СГУ. 2007. – 66–69 с. [Электронный ресурс] – Режим доступа: http://is.ifmo.ru/works/\_2007\_10\_05\_mps\_textual\_language.pdf
34. Ахо А. Компиляторы: принципы, технологии и инструментарий, 2-е издание / А. Ахо, М. Лам, Р. Сети, Д. Уильман – ООО «И. Д. Вильямс», 2011. – 1184 с: ил.
35. Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто A. А. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6,с. 12–17. [Электронный ресурс] – Режим доступа: http://is.ifmo.ru/works/UML-SWITCH-Eclipse.pdf
36. Гуров В. С., Нарвский А. С., Шалыто А. А. Исполняемый UML из России [Электронный ресурс] – Режим доступа: http://is.ifmo.ru/works/\_umlrus.pdf
37. Гуров В. С., Мазин М. А., Шалыто А. А. Операционная семантика UML-диаграмм состояний в программном пакете UniMod // Труды XII Всероссийской научно-методической конференции «Телематика-2005». СПб.:СПбГУ ИТМО. Т.1, с. 74–76. [Электронный ресурс] – Режим доступа: http://tm.ifmo.ru/tm2005/src/224as.pdf
38. Лифшиц Ю. Верификация программ и темпоральные логики Лекция №3 курса "Современные задачи теоритической информатики" [Электронный ресурс] – Режим доступа: http://yury.name/modern/03.pdf
39. Проверка моделей [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Model\_checking
40. Карпов Ю. Г. Model Checking. Верификация параллельных и распределенных программ / Ю.Г. Карпов – СПб.: БХВ-Петербург, 2010.– 552 с.
41. Holzman G. J. Design and validation of computer protocols / G. J. Holzman – NJ: Pentice Hall, 1991. – 544 c.
42. Holzman G. J. The model checker SPIN [Электронный ресурс] – Режим доступа: http://spinroot.com/spin/Doc/ieee97.pdf
43. SPIN home page [Электронный ресурс] – Режим доступа: http://SPINroot.com
44. Promela [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Promela
45. Linear Temporal Logic [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Linear\_temporal\_logic
46. Модель Крипке [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Kripke\_structure\_(model\_checking)
47. Büchi automaton [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/B%C3%BCchi\_automaton
48. Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs [Электронный ресурс] – Режим доступа: http://www.cs.virginia.edu/~weimer/2008-615/reading/DijkstraGC.pdf
49. Кюзара В.Е. Реализация системы проверки моделей раскрашенных сетей Петри с использованием разверток [Электронный ресурс] – Режим доступа: http://www.iis.nsk.su/files/preprints/094.pdf
50. Vardy M.Y., Wolper P. An automata-theoretic approach to automatic program verification [Электронный ресурс] – Режим доступа: http://spinroot.com/spin/Doc/lics86.pdf
51. GNU Compiler Collection [Электронный ресурс] – Режим доступа: http://gcc.gnu.org/
52. Сайт Java [Электронный ресурс] – Режим доступа: http://java.com/en/
53. Aвтоматическая коробка передач [Электронный ресурс] – Режим доступа: http://en.wikipedia.org/wiki/Automatic\_transmission
54. JavaCC [Электронный ресурс]: - Режим доступа: http://javacc.java.net/
55. GNU bison [Электронный ресурс]: - Режим доступа: http://www.gnu.org/software/bison/
56. Flex [Электронный ресурс]: - Режим доступа: http://flex.sourceforge.net/
57. Yacc [Электронный ресурс]: - Режим доступа: http://dinosaur.compilertools.net/
58. Сосо/R [Электронный ресурс]: - Режим доступа: http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
59. Крючкова Е. Н. Основы теории языков программирования и методов трансляции. Анализ: Учебное пособие / Алт. Госуд. Технич ун-т им. И. И. Ползунова. Барнаул, 2010. – 244 с.
60. ANTLR [Электронный ресурс]: - Режим доступа: http://www.antlr.org/
61. Apache Maven [Электронный ресурс]: - Режим доступа: http://maven.apache.org/
62. Расширенная форма Бэкуса — Наура [Электронный ресурс]: - Режим доступа: http://en.wikipedia.org/wiki/Extended\_Backus%E2%80%93Naur\_Form
63. Абстрактное синтаксическое дерево [Электронный ресурс]: - Режим доступа: http://en.wikipedia.org/wiki/Abstract\_syntax\_tree
64. ANTLR Tree construction [Электронный ресурс]: - Режим доступа: http://www.antlr.org/wiki/display/ANTLR3/Tree+construction
65. Граф потока исполнения [Электронный ресурс]: - Режим доступа: http://en.wikipedia.org/wiki/Control\_flow\_graph
66. М. А. Лукин Верификация автоматных программ [Электронный ресурс]: - Режим доступа: http://is.ifmo.ru/papers/\_lukin\_bachelor.pdf
67. Описание Промелы и некоторых примеров программ [Электронный ресурс]: - Режим доступа: http://sevik.ru/mstu/docs/promela-rtf.zip
68. Xtext [Электронный ресурс]: - Режим доступа: http://www.eclipse.org/Xtext/
69. Ю.Е. Шатилина Реализация системы проектирования и тестирования каркасов программных продуктов / Научно-образовательный журнал АлтГТУ "Горизонты образования", 2012, вып.14, Приложение: 9-я Всероссийская научно-техническая конференция «Наука и молодежь-2012». Секция "Информационные технологии". Подсекция "Программное обеспечение вычислительной техники и автоматизированных систем" сс. 58-60 [Электронный ресурс]: - Режим доступа: http://edu.secna.ru/media/f/povt2012.pdf