Содержание

[Введение 3](#_Toc322990629)

[1 Прототипирование архитектуры ПО 4](#_Toc322990631)

[1.1 Прототипирование ПО 4](#_Toc322990632)

[1.2 Понятие прототипа архитектуры ПО 7](#_Toc322990633)

[1.3 Использование прототипов архитектуры ПО 8](#_Toc322990634)

[1.4 Требования к прототипам архитектуры ПО 13](#_Toc322990635)

[1.5 Инструменты прототипирования архитектуры ПО 14](#_Toc322990636)

[1.6 Проверка правильности прототипов 15](#_Toc322990637)

[1.7 Выводы 16](#_Toc322990638)

[2 Модель системы прототипирования 17](#_Toc322990639)

[2.1 Общие положения 17](#_Toc322990640)

[2.2 Прототип архитектуры 17](#_Toc322990641)

[2.3 Модель прототипа архитектуры 17](#_Toc322990642)

[2.4 Описание модели прототипа 18](#_Toc322990643)

[2.5 Внутреннее представление модели прототипа 18](#_Toc322990644)

[2.6 Оптимизация модели прототипа 21](#_Toc322990645)

[2.7 Анализ модели прототипа 21](#_Toc322990646)

[2.8 Верификация модели прототипа 21](#_Toc322990647)

[2.9 Выводы 22](#_Toc322990648)

[3 Реализация системы 23](#_Toc322990649)

[4 Организационно-экономический раздел 24](#_Toc322990650)

[5 Охрана труда и окружающей среды 25](#_Toc322990651)

[Заключение 26](#_Toc322990652)

[Список использованных источников 27](#_Toc322990653)

[Приложение А Задание на дипломное проектирование 29](#_Toc322990654)

[Приложение Б Руководство пользователя 31](#_Toc322990655)

[Приложение В Код программы 32](#_Toc322990656)

# Введение

Современные методы и практики быстрой разработки программного обеспечения нацелены на минимизацию рисков, путем сведения процесса к серии коротких циклов, каждый их которых пердставляет 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 его разработчикам. Кроме того проектирование является едва ли не самой крупной частью процесса разработки и в значительной степени влияет на успешность всего процесса разработки. Кроме того процесс проектирования является своего рода центральной частью разработки: до него происходит этап выработки требований к разрабатываемому программному обеспечению и начальной разработки его архитектуры, после него – тестирование и доработка 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.

Идеальный программный проект до начала конструирования проходит стадии тщательной выработки требований и проектирования архитектуры [1]. 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:

* детальное проектирование – 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 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 основными стадиями можно отнести следующие:

* определение начальных требовании,
* 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. 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 – это упрощенное представление желамой архитектуры, без учета деталей и аспектов реализации. Протипы 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 еще 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 применяемый выход из этой ситуации – это составление предельно подробных спецификаций системы. Написание большого количества документации, которые будут четко 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» [2]. 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 путем параллельного наращивания каждого компонента «скелета». Этот «скелет» также принято называть программой трассировки. Она содержит всю проверку ошибок, документацию и структурирование, которые имеющиются в любом фрагменте рабочей программы. Единственное ее 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 проверить, 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 идет 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 тяжеловестный технический подход, при котором вся разрабатываемая система делится на модули, разработка которых ведется в вакууме. Модули объединены в подсистемы, которые в дальнейшем тоже подлежат объединению, пока в конечном итоге не получится завершенное приложение. Оно то и может быть представлено конечному пользователю и протестировано.

Технология программы трассировки имеет следующие приемущества:

* *можно предоставить пользователям некий работающий вариант системы еще 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 модулми системы и есть их реализация, то команде разработчиков не придется 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, вследствии 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 объему очень мала, то можно избежать создания больших монолитных программных блоков;

Как известно, трассирующие пули только показывают, куда вы попали. Но не обязательно вы попали в цель. С учетом того, куда вы попали, вы затем корректируете прицел, пока не добьетесь 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 ситации, 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*). Такие методологии разработки пердполагают сведение разработки программного обеспечения к серии коротких циклов с целью минимизации рисков. Такие короткие циклы еще 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 метрико 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. Среди методологии разработки, которые придерживаются манифеста можно выделить следующие методологий:

* *Agile Modeling* - набор понятий и приемов, 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, развертывания и сопровождения системы. Однако включает в себя проверку модели кодом;
* *Agile Unified Process (AUP)* упрощенная 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 ее функциональная часть;
* *Extreme Programming* – процесс разработки, основанный на 12 основных приемах, которые объединены в 4 группы: короткий цикл обратной связи, непрерывный процесс разработки, разделяемое всеми понимание и социальная защищенность программиста;
* *Scrum* - 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 дает 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 определенных характеристик и аспектов конечной версии системы. При создании истинного прототипа будет отброшено все то, что критиковалось при тестировании прототипа, и он будет переписан с учетом 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 однаразовую программу, в отличие от программ, полученных одним из вышеуказанных методов – простых, но завершенных, 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 определенных 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 приемлимо, можно использовать фиктивные данные,
* завершенность – прототип может функционировать лишь в ограниченном смысле, возможно лишь с одним заданным фрагментом данных и одним пунктом меню,
* надежность – процедура проверки ошибок, вероятно, будет неполной или будет отсутствовать полностью. При отклонении от определенного пути прототип может выйти из строя,
* стиль – прототип программы не имеет большого значения для комментариев или документации.

Готовый прототип архитектуры системы должен дать ответы на многие вопросы. Например:

* четко ли определены обязанности основных компонентов,
* являются ли эти обязанности приемлимыми для компонентов,
* четко ли определена совместная работа основных компонентов,
* сведено ли к минимуму связывание между компонентами,
* можно ли выделить потенциальные источники дублиования,
* можно ли применять определения интерфейсов и ограничения,
* обладает ли каждый из модулей путем 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 создается с целью моделирования рассматриваемой системы в целом, то полученный прототип есть не что иное, как одноразовая программа, необходимая для того, чтобы получить ответы на определенный ряд вопросов. В прототипах опущены ненужные детали и подробности, что позволяет в центре рассмотрения иметь лишь определенные 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 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 соеденить 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.

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* – доказательство соответствия текста программной системы ее формальной спецификации.

В формальной верификации выделяют несколько основных направлении:

* *доказательная верификация*;
* *проверка модели* системы *(Model Checking)*. 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). Результатом верификации модели может быть подтверждение того, что модель удовлетворяет предъявленным тербованиям, либо контрпример. Построение контрпримера требует выявления причины некорректности.

## 1.7 Выводы

Согласно приведенным 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.

Автором предлагается проект системы описания и тестирования прототипов программных архитектур, которая позволяет обеспечить выполнение перечисленных требований. Основополагающая идея предлагаемого 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.

В некотором смысле, можно считать модель рассмотренной системы моделью оптимизирующего компилятора, включающего дополнительную стадию верификации.

Коротко объяснить как описывается система, стадии – коротко о том что будет дальше.

## 2.2 Прототип архитектуры

Описать в терминах ООП (суперпозиция, агрегация, композиция)

Минимальный набор сущностей предметной области

Под *прототипом архитектуры* далее будем понимать абстракцию, характеризующуюся:

1. основными данными об элементах архитектуры;
2. данными о связи между элементами архитектуры;
3. отсутствием излишних деталей;
4. реализацией, представляющей собой код на унифицированном языке.

## 2.3 Модель прототипа архитектуры

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

*M = (P, S, R, U)*, где

P – прототип архитектуры,

S – множество состояний прототипа,

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 путем выделения состоянии, сбора спецификации требовании по работе прототипа, а также формализации всех возможных сценариев использования прототипа.

Состояние в работе прототипа можно определить как особый момент в работе прототипа архитектуры, который объедининяет 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 определенный смысл и качественно отличается от всех других состоянии, и кроме того однозначно определяет действия, которые могут соверщаться в этом состоянии.

Набор спецификации модели прототипа представляет собой список свойств, которыми должен обладать прототип в том или ином состоянии. Иными словами, спецификации требовании отражают желаемое поведение прототипа архитектуры. Спецификации обычно удобно формулируются на языке темпоральной логики. Темпоральная логика представляет собой логику, специализированную для учета причинно-следственных связей во времени. С помощью такой логики можно легко установить последовательность и взаимосвязь явлений по временной шкале.

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

Такое описание модели прототипа позволяет значительно упростить подход к верификации исходного прототипа, а так же сделать этот процесс более строгим и формальным. В методах верификации систем в качестве моделей обычно используют так называемую модель Крипке. Эту модель формально можно описать над множеством атомарных высказываний AP (булевых выражений над множеством переменных, констант и предикатных символов) с помощью четверки вида:

*M = (S, S0, R, L)*, в которой

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 ориентированным графом, вершины которого описывают достижимые состояния, а ребра - переходы из состояния в состояние. Функция пометок сопоставляет каждой вершине множество свойств, которые выполняются в соответствующем состоянии.

## 2.4 Описание модели прототипа

При описании прототипа архитектуры используется унифицированный язык (Domain-specific language).

Язык, Грамматика,

## 2.5 Внутреннее представление модели прототипа

? убрать

Внутреннее представление прототипа используется на промежуточных этапах работы системы с целью получения информации о работоспособности исходного прототипа. Таким образом, от того насколько удачно будет выбрано внутренне представление напрямую зависит результат работы всей системы. Используемое внутреннее представление должно позволять выполнить все необходимые проверки и оптимизации, а кроме того еще проводить тестирование прототипа.

Внутреннее представление прототипа удобно представить воспользуясь парадигмой автоматного программирования. Автоматное программирование («программирование от состояний» или «программирование с явням выделением состояний») представляет собой подход к разработке программного обеспечения, в основе которого лежит расширенная модель конечных автоматов [5].

Известно, что конечный автомат можно описать шестеркой вида

*A = (K, ∑, δ, p0, F)*, где

K – конечное множество состояний,

∑ - алфавит,

δ - функция переходов,

p0 – начальное состояние, p0 K,

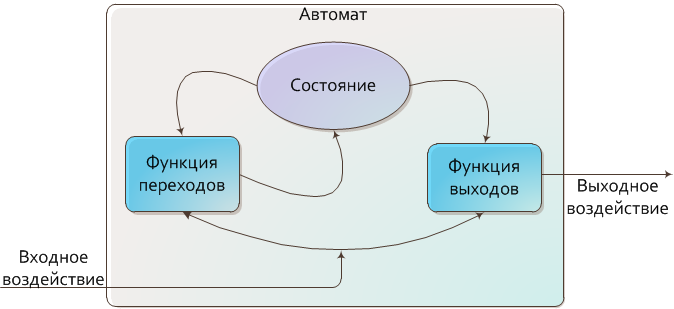
F – множество заключительных состояний, F ⊆ K.

Конечный автомат можно определить как формальную систему через состояния, символы, которые пишутся или читаются с ленты или нескольких лент, и набора команд [6].

Одной из базовых характеристик конечного автомата согласно автоматному программированию является его *состояние*. Основное свойство состояния конечного автомата в момент времени t0 заключается в разделении между будущим (t>t0) и прошлым (t<t0) в том смысле, что текущее состояние включает в себя информацию о прошлых событиях, которая необходима для определения реакции на любое входное воздействие, формируемое в момент времени t0. Таким образом, состояние – это особая характеристика, объединяющая в неявной форме все входные воздействия прошлого, а также влияющая на реакцию в текущий момент времени. А значит, реакция зависит от входного воздействия и текущего состояния конечного автомата. Каждое состояние конечного автомата имеет вполне определенный смысл и качественно отличается от всех других состоянии, и кроме того однозначно определяет действия, которые совершает сущность.

Под *входным воздействием* далее будем понимать вектор, среди компонетов которого можно выделить события и входные переменные в зависимости от смысла и механизмов формирования. Результатом объединения конечного множества состояний с конечным множеством входных воздействий является конечный автомат без выходов. Такой автомат определенным образом изменяет текущее состояние в результате реакции на входные воздействия. Правила, по которым происходит смена состояний, называют *функцией переходов* автомата.

При комбинировании понятия автомата без выходов с понятием «входного воздействия» получим *конечный автомат* (см. Рисунок 2.1). Автомат, соответствующий такому описанию, реагирует на входные воздействия сменой состояние и формированием определенных значений на выходах. Правила, по которым происходит формирование значений выходных воздействии, называют *функциями выходов* автомата.

Рисунок 2.1 – Конечный автомат

Существует несколько способов описания конечных автоматов:

* *граф переходов* (или диаграмма переходов) – графическое представление множества состояний и функций переходов. Предсталяет собой ориентированный граф, вершинам которого соответствуют управляющие состояния конечного автомата, дугам – переходы из одного состояния в другое. Дуга такого графа помечается условием перехода и выходным воздействием на переходе. Условие перехода принято отделять от выходного воздействия горизонтальной чертой (см. Рисунок 2.2) ;
* *таблица переходов* – табличное представление функции перехода конечного автомата δ. Как правило, в таблице переходов каждой строке соответствует одно состояние, а столбцу — один допустимый входной символ. Ячейка на пересечении строки и столбца хранит действие, которое должен выполнить автомат, если в ситуации, когда он находился в данном состоянии, на входе он получил данный символ.

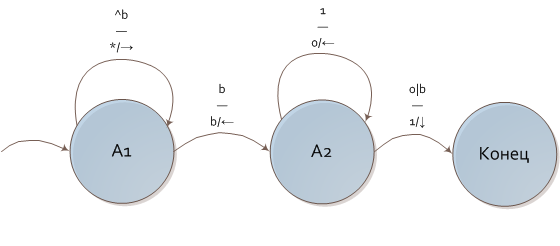


Рисунок 2.2 – Пример конечного автомата, представленного графом переходов

## 2.6 Оптимизация модели прототипа

Оптимизации Крипке?

Внутреннее представление прототипа в виде конечного автомата имеет много приемуществ, среди которых можно выделить возможность проанализировать и модифицировать это представление с целью упрощения.

выполняет ряд оптимизации над внутренним представлением, например, детерминизацию и минимизацию графа переходов для упрощения процесса тестирования;

## 2.7 Анализ модели прототипа

* осуществляет поиск шаблонов проектирования, типичных ошибок в построении архитектуры, формирует отчет и рекомендации наоснове полученных данных;

## 2.8 Верификация модели прототипа

Процесс тестирования прототипа позволяет установить факт работоспособности исходного прототипа архитектуры, а так же факт его соответствия предъявленным требованиям.

Переходя к внутреннему представлению 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 событии, в которых пребывал автомат, обработанные события, значения 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 состяния 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 учета семантики входных и выходных переменных). Дополнительным приемуществом при проверке может служить наличие спецификации.

# 3 Реализация системы

# 4 Организационно-экономический раздел

# 5 Охрана труда и окружающей среды

# Заключение

# 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. Хант Э. Программист-прагматик. Путь от подмастерья к мастеру / Э. Хант, Д. Томас - СПб.: Питер, 2010. – 830 с.
3. 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
4. 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.
5. Поликарпова Н. Автоматное программирование / Н. Поликарпова, А. Шалыто – СПб.: Питер, 2011.- 176 с: ил.
6. Крючкова Е. Основы математической логики и теоpии алгоpитмов: Учебное пособие / Е. Крючкова – Алт. госуд. технич. ун-т им. И.И.Ползунова. Барнаул, 2010. — 277с.
7. Хопкрофт Д. Введение в теорию автоматов, языков и вычислений / Д. Хапкрофт, Р. Мотвани, Д. Ульман – М.:Вильямс, 2002.
8. Кузьмин Е. В., Соколов В. А. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем / ЯрГУ. Ярославль. 2007. № 1. С. 3–14. http://is.ifmo.ru, раздел «Верификация».
9. Finite state machine [Электронный ресурс] / Режим доступа: <http://en.wikipedia.org/wiki/Finite_state_machine>
10. Mealy machine [Электронный ресурс] / Режим доступа: <http://en.wikipedia.org/wiki/Mealy_machine>
11. Switch-technology [Электронный ресурс] / Режим доступа: http://en.wikipedia.org/wiki/Switch-technology
12. Roux C., Encrenaz E. CTL May Be Ambiguous when Model Checking Moore Machines. UPMC LIP6 ASIM. CHARME, 2003. <http://sed.free.fr/cr/charme2003presentation.pdf>
13. Clarce E. M., Emerson E. A., Sistla A. P. Automatic Verification of Finite State Concurrent Using Temporal Logic Specifications //ACM Transactions on Programming Languages and Systems (TOPLAS). 1986. Vol. 8. N 2. P. 244–263.
14. Сайт Java [Электронный ресурс] / Режим доступа: <http://java.com/en/>
15. NetBeans Project [Электронный ресурс] / Режим доступа: <http://www.netbeans.org/>
16. Шилдт, Г. Искусство программирования на Java / Г. Шилдт, Д. Холмс – Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 336 с.
17. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – СПб.: Питер, 2009. – 366 с.
18. Головач, В.В. Дизайн пользовательского интерфейса / В.В. Головач – 2000. – 141 с.
19. Oracle tutorial for swing [Электронный ресурс] / Режим доступа: <http://download.oracle.com/javase/tutorial/uiswing/>
20. JTattoo official website [Электронный ресурс] / Режим доступа: <http://www.jtattoo.net/>

# Приложение А Задание на дипломное проектирование

# Приложение Б Руководство пользователя

# Приложение В Код программы