1. Идентификация, анализ и детализация **объекта изучения**, представленного соответствующими **процессами**  системного и программного контекстов Опорной (ссылочной) модели процессов ЖЦ ПО (стандарт ISO/IEC/IEEE 12207:2008 [1]), имеющими отношение к данной теме: ***«Обеспечение (гарантирование) качества ПО» (7.2.3), «Квалификационное тестирование ПО в системном и программном контекстах» (6.4.6, 7.1.7), «Верификация ПО» (7.2.4), «Валидация ПО» (7.2.5), «Гарантирование (обеспечение) качества ПО» (7.2.3).***

**Примечание:** Предлагаемый состав анализируемых процессов может быть **обосновано**  изменен (дополнен, сокращен) исполнителем по соображениям их значения («веса») для раскрытия выбранной темы.

1. Раскрытие спецификации (детализации структуры) анализируемых процессов в соответствии с ISO/IEC/IEEE 12207:2008 [1] в терминах: **контекст процесса, группа процессов, процесс в группе – объект изучения выбранной темы, назначение (цель процесса), выходной продукт, действия, задачи.**
2. Определение для выбранного состава процессов на основе аналитических обзоров рекомендованной литературы предметной области выбранной темы, представленной соответствующей методологией и технологиями в терминах: **абстракции, модели, методы, меры и метрики, инструментальные средства и среды, информационные технологии, руководства и стандарты.**
3. Классификация методов верификации ПО. Место и роль статического анализа.
4. Цели, задачи, возможности, ограничения статического анализа исходных кодов ПО.
5. Адаптация к ЯВУ;
6. Проблема отчуждаемости критического ПО.
7. Теоретические основы анализа текстов ПО на языках высокого уровня.
8. Синтаксический разбор текста ПО. Парсинг. Формальный анализ  исходных текстов ПО.
9. Перспективы использования статического анализа текстов ПО.
10. Инструментальные средства и среды. Возможности Ограничения.
11. Формальные методы анализа и верификации критического ПО на основе статического анализа.
12. Полимодельная Model-checking верификация с использованием инварианто-ориентированных моделей критического ПО. На основе статического анализа исходных текстов ПО.

Введение  
Прогресс технологий разработки программного обеспечения (ПО) в последние десятилетия значительно увеличил производительность программистов в терминах количества кода, создаваемого ими в единицу времени. Это проявляется, в частности, в увеличении размеров наиболее сложных программных систем, разрабатываемых сейчас, до десятков миллионов строк кода. Однако качество программ при этом заметным образом не изменилось — среднее количество ошибок на тысячу строк кода, еще не прошедшего тестирование, по-прежнему колеблется в пределах 10-50. Таким образом, совершенствование методов разработки ПО, давая возможность создавать все более сложные системы, необходимые современнойэкономике, науке и государственным организациям, парадоксальным образом лишь увеличивает количество дефектов в них и связанные с этим риски.  
Борьба с дефектами и ошибками в программном обеспечении ведется при помощи его верификации. В ходе ее выполнения проверяется взаимная согласованность всех артефактов разработки — проектной и пользовательской документации, исходного кода, конфигураций развертывания, — а также их соответствие требованиям к данной системе и нормам применимых к ней стандартов. Методы верификации ПО также активно развиваются, однако их прогресс менее заметен. Поэтому предельная сложность ПО, которое можно сделать надежно и корректно работающим, существенно меньше сложности систем, востребованных современным обществом.  
Различные методы проведения верификации ПО можно (больше по историческим, чем содержательным причинам) разделить на формальные методы, использующие строгий анализ математических моделей проверяемых артефактов и требуемых свойств;  методы статического анализа, в ходе которых возможные ошибки ищутся без исполнения проверяемого ПО;  методы динамического анализа, проводящие проверку реального поведения проверяемой системы в рамках некоторых сценариев ее работы; и экспертизу (review, inspection), выполняемую людьми на основе их знаний и опыта. Все эти методы имеют разные достоинства и недостатки, различные области применимости, и эффективность их применения сильно отличается в разных контекстах. Но полноценная верификация крупномасштабных сложных систем невозможна без совместного использования всех этих методов, поскольку только их сочетание позволяет преодолеть недостатки каждого.  При использовании формальных методов задачи тестирования могут быть в значительной степени решены посредством инспектирования — статического анализа исходных текстов программ с целью верификации соответствия формальной модели. Инспектированию следует подвергать также модели анализа и архитектуры, причем непосредственно при их создании в ходе соответствующих фаз технологического цикла, что позволяет минимизироватьобщее количество дефектов в разрабатываемой системе. Есть автоматические инструменты статического анализа, помогающие выявлять фрагменты моделей, активизация которых способна вызвать нежелательные эффекты.[1]

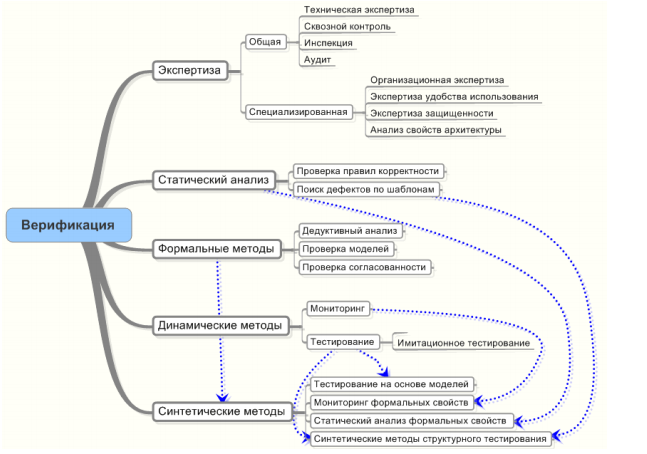
1.Идентификация, анализ и детализация объекта изучения, представленного соответствующими процессами системного и программного контекстов Опорной (ссылочной) модели процессов ЖЦ ПО   
Под понятием «программное обеспечение, критичное с точки зрения безопасности» (Safety-Critical Software – для краткости будем называть его «критичное программное обеспечение») обычно понимают такое программное обеспечение, которое влияет на поведение систем, сбой которых может повлечь риск для человеческих жизней. Иногда этим же термином называют программы, разработанные в соответствии со специальными стандартами, принятыми для критически важных областей.[1]  
Главным препятствием, стоящим на пути применения «коробочных» программных продуктов в критических областях, является недостаточная информация о «родословной» изделий, т.е. о процессах разработки, тестирования, верификации и управления качеством и т.п.. Эту проблему в значительной мере решает как лицензирование деятельности предприятий, так и сертификация соответствующей продукции на соответствие определенным требованиям, сформулированным, как правило, в виде стандартов или иных руководящих документов. К функциональной безопасности программного обеспечения имеет отношение ряд международных и национальных, общих и отраслевых стандартов, регламентирующих жизненный цикл технических систем.  
Опорная модель подходов к качеству предназначена для описания, сравнения и анализа подходов к менеджменту качества и подтверждения качества. Сама модель, таким образом, не является моделью менеджмента качества или подтверждения качества – это подход для описания подходов к качеству. Она служит для сравнения различных существующих стандартов и для гармонизации их с целью построения общей модели качества. Для лучшего понимания в стандарт включены ряд приложений, описывающих лучшие практики ряда страни критерии, которые можно предложить в качестве опорных критериев для анализа и оценки обучающих ресурсов и сценариев.   
Эти критерии также, сами по себе, не являются подходом к оценке качества,   
но могут быть использованы в качестве модели сравнения различных подходов к подтверждению и оценке качества.  
Группы процессов жизненного цикла включают в себя:  
процессы соглашения — 2;  
процессы организационного обеспечения проекта — 5;  
процессы проекта — 7;  
технические процессы — 11;  
процессы реализации программных средств — 7;  
процессы поддержки программных средств — 8;  
процессы повторного применения программных средств — 3.  
Процессы соглашения  
Поставка  
Приобретение  
Процессы организационного обеспечения проекта  
Процесс менеджмента модели жизненного цикла;  
Процесс менеджмента инфраструктуры;  
Процесс менеджмента портфеля проектов;  
Процесс менеджмента людских ресурсов;  
Процесс менеджмента качества.  
Процессы проекта  
Процессы менеджмента проекта  
процесс планирования проекта;  
процесс управления и оценки проекта.  
Процессы поддержки проекта  
процесс менеджмента решений;  
процесс менеджмента рисков;  
процесс менеджмента конфигурации;  
процесс менеджмента информации;  
процесс измерений.  
Технические процессы  
Определение требований правообладателей  
Анализ системных требований  
Проектирование архитектуры системы  
Процесс реализации  
Процесс комплексирования системы  
Процесс квалификационного тестирования системы  
Процесс инсталляции программных средств  
Процесс поддержки приемки программных средств  
Процесс функционирования программных средств  
Процесс сопровождения программных средств  
Процесс изъятия из обращения программных средств  
Процессы реализации программных средств  
Процесс анализа требований к программным средствам;  
Процесс проектирования архитектуры программных средств;  
Процесс детального проектирования программных средств;  
Процесс конструирования программных средств;  
Процесс комплексирования программных средств;  
Процесс квалификационного тестирования программных средств  
Процессы поддержки программных средств  
Процесс менеджментадокументации программных средств;  
Процесс менеджмента конфигурации программных средств;  
Процесс обеспечения гарантии качества программных средств;  
Процесс верификации программных средств;  
Процесс валидации программных средств;  
Процесс ревизии программных средств;  
Процесс аудита программных средств;  
Процесс решения проблем в программных средствах.  
Процессы повторного применения программных средств  
Процесс проектирования доменов;  
Процесс менеджмента повторного применения активов;  
Процесс менеджмента повторного применения программ.  
Другим способом деления рассмотрим процессы приобретения, поставки, разработки, эксплуатации и сопровождения.  
Приобретение   
Процесс приобретения (как его называют в ГОСТ – “заказа”) определяет работы и задачи заказчика, приобретающего программное обеспечение или услуги, связанные с ПО, на основе контрактных отношений. Процесс приобретения состоит из следующих работ (названия ГОСТ 12207 даны в скобках, если предлагают другой перевод названий работ оригинального стандарта):  
Inititation – инициирование (подготовка)  
Request-for-proposal preparation – подготовка запроса на предложение (подготовка заявки на подряд)  
Contract preparation and update –подготовка и корректировка договора  
Supplier monitoring – мониторинг поставщика (надзор за поставщиком)  
Acceptance and completion – приемка и завершение (приемка и закрытие договора)  
Все работы проводятся в рамках проектного подхода.  
Поставка   
Процесс поставки, в свою очередь, определяет работы и задачи поставщика. Работы также проводятся с использованием проектного подхода. Процесс включает следующие работы:  
Inititation – инициирование (подготовка)  
Preparation of response – подготовка предложения (подготовка ответа)  
Contract – разработка контракта (подготовка договора)  
Planning - планирование  
Execution and control – выполнение и контроль  
Review and evaluation –проверка и оценка  
Delivery and completion – поставка и завершение (поставка и закрытие договора)  
Разработка   
Процесс разработки определяет работы и задачи разработчика. Процесс состоит изследующих работ:  
Process implementation – определение процесса (подготовка процесса)  
System requirements analysis – анализ системных требований (анализ требований к системе)  
System design – проектирование системы (проектирование системной архитектуры)  
Software requirements analysis – анализ программных требований (анализ требований к программным средствам)  
Software architectural design – проектирование программной архитектуры  
Software detailed design – детальное проектирование программной системы (техническое проектирование программных средств)  
Software coding and testing – кодирование и тестирование (программирование и тестирование программных средств)  
Software integration – интеграция программной системы (сборка программных средств)  
Software qualification testing – квалификационные испытания программных средств  
System integration – интеграция системы в целом (сборка системы)  
System qualification testing – квалификационные испытания системы  
Software installation – установка (ввод в действие)  
Software acceptance support – обеспечение приемки программных средств  
Стандарт отмечает, что работы проводятся с использованием проектного подхода и могут пересекаться по времени, т.е. проводиться одновременно или с наложением, а  также могут предполагать рекурсию и разбиение на итерации.  
Эксплуатация   
Процесс разработки определяет работы и задачи оператора службы поддержки. Процесс включает следующие работы:  
Process implementation – определение процесса (подготовка процесса)  
Operational testing – операционное тестирование (эксплуатационные испытания)  
System operation         – эксплуатация системы  
User support – поддержка пользователя  
Сопровождение   
Процесс разработки определяет работы и задачи, проводимые специалистами службы сопровождения. Процесс включает следующие работы:  
Process implementation – определение процесса (подготовка процесса)  
Problem and modification analysis – анализ проблем и изменений  
Modification implementation – внесение изменений  
Maintenance review/acceptance – проверка и приемка при сопровождении  
Migration –миграция (перенос)  
Software retirement – вывод программной системы из эксплуатации (снятие с эксплуатации)  
Важно понимать, что стандарт 12207 не определяет последовательность и разбиение выполнения процессов во времени, адресуя этот вопрос также работам по адаптации стандарта к конкретным условиям и окружению и применению выбранных моделей, практик, техник  и т.п.[1]  
Вывод: Опорная модель подходов к качеству предназначена для описания, сравнения и анализа подходов к менеджменту качества и подтверждения качества. Она служит для сравнения различных существующих стандартов и для гармонизации их с целью построения общей модели качества.

2.

Раскрытие спецификации (детализации структуры) анализируемых процессов в соответствии с ISO/IEC/IEEE 12207:2008 в терминах: контекст процесса, группа процессов, процесс в группе – объект изучения выбранной темы, назначение (цель процесса), выходной продукт, действия, задачи.  
Контекстом процесса является его состояние, определяемое текстом, значениями глобальных переменных пользователя и информационными структурами, значениями используемых машинных регистров, значениями, хранимыми в позиции таблицы процессов и в адресном пространстве задачи, а также содержимым стеков задачи и ядра, относящихся к данному процессу. Текст операций системы и ее глобальные информационные структуры совместно используются всеми процессами, но не являются составной частью контекста процесса.  
Говорят, что при запуске процесса система исполняется в контексте процесса. Когда ядро системы решает запустить другой процесс, оно выполняет переключение контекста с тем, чтобы система исполнялась в контексте другого процесса. Ядро осуществляет переключение контекста только при определенных условиях, что мы увидим в дальнейшем. Выполняя переключение контекста, ядро сохраняет информацию, достаточную для того, чтобы позднее переключиться вновь на первый процесс и возобновить его выполнение. Аналогичным образом, при переходе из режима задачи в режим ядра, ядро системы сохраняет информацию, достаточную для того, чтобы позднеевернуться в режим задачи и продолжить выполнение с прерванного места. Однако, переход из режима задачи в режим ядра является сменой режима, но не переключением контекста. Ядро обрабатывает прерывания в контексте прерванного процесса, пусть даже оно и не вызывало никакого прерывания. Прерванный процесс мог при этом выполняться как в режиме задачи, так и в режиме ядра. Ядро сохраняет информацию, достаточную для того, чтобы можно было позже возобновить выполнение прерванного процесса, и обрабатывает прерывание в режиме ядра. Ядро не порождает и не планирует порождение какого-то особого процесса по обработке прерываний.  
Каждому процессу соответствует контекст, в котором он выполняется. Этот контекст включает содержимое пользовательского адресного пространства - пользовательский контекст (т.е. содержимое сегментов программного кода, данных, стека, разделяемых сегментов и сегментов файлов, отображаемых в виртуальную память), содержимое аппаратных регистров - регистровый контекст (таких, как регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистров общего назначения), а также структуры данных ядра (контекст системного уровня), связанные с этим процессом. Контекст процесса системного уровня в ОС UNIX состоит из "статической" и "динамических" частей. У каждого процесса имеется одна статическая часть контекста системного уровня и переменное число динамических частей.   
Статическая часть контекста процесса системного уровня включает следующее:   
1. Описатель процесса, т.е. элемент таблицы описателей существующих в системе процессов. Описатель процесса включает, в частности, следующую информацию:   
состояние процесса;   
физический адрес в основной или внешней памяти u-области процесса;   
идентификаторы пользователя, от имени которого запущен процесс;   
идентификатор процесса;   
прочую информацию, связанную с управлением процессом.   
2. U-область (u-area), индивидуальная для каждого процесса область пространства ядра, обладающая тем свойством, что хотя u-область каждого процесса располагается вотдельном месте физической памяти, u-области всех процессов имеют один и тот же виртуальный адрес в адресном пространстве ядра. Именно это означает, что какая бы программа ядра не выполнялась, она всегда выполняется как ядерная часть некоторого пользовательского процесса, и именно того процесса, u-область которого является "видимой" для ядра в данный момент времени. U-область процесса содержит:   
указатель на описатель процесса;   
идентификаторы пользователя;   
счетчик времени, которое процесс реально выполнялся (т.е. занимал процессор) в режиме пользователя и режиме ядра;   
параметры системного вызова;   
результаты системного вызова;   
таблица дескрипторов открытых файлов;   
предельные размеры адресного пространства процесса;   
предельные размеры файла, в который процесс может писать;   
и т.д.   
Динамическая часть контекста процесса - это один или несколько стеков, которые используются процессом при его выполнении в режиме ядра. Число ядерных стеков процесса соответствует числу уровней прерывания, поддерживаемых конкретной аппаратурой. [2]  
Цель процесса (process purpose) - цель высокого уровня выполнения процесса и вероятные выходы эффективной реализации процесса.  
Необходимо, чтобы реализация процесса обеспечивала ощутимую пользу правообладателям.  
Выход процесса (process outcome) - наблюдаемый результат успешного достижения цели процесса. Формулировка выхода процесса описывает один из следующих результатов:  
- изготовление какого-либо артефакта;  
- существенное изменение состояния;  
- удовлетворение заданных ограничений, например требований, конечных целей и т.п.  
Продукт (product) - результат процесса. [3]  
Группы процессов   
Если большие или сложные проекты могут быть разбиты на отдельные фазы или подпроекты, например, анализ осуществимости, разработка идеи,   
проектирование, создание прототипа, производство, испытание и т.д., то все группы процессов обычно будут применяться к каждой фазе или подпроекту.   
Пять групп процессов таковы:   
Группа процессов инициации. Определяет и авторизует проект или фазу проекта.   
Группапроцессов планирования. Определяет и уточняет цели и планирует действия, необходимые для достижения целей и содержания, ради которых был предпринят проект.   
Группа процессов исполнения. Объединяет человеческие и другие ресурсы для выполнения плана управления проектом данного проекта.   
Группа процессов мониторинга и управления. Регулярно оценивает прогресс проекта и осуществляет мониторинг, чтобы обнаружить отклонения от плана управления проектом, и, в случае необходимости, провести корректирующие действия для достижения целей проекта.   
Группа завершающих процессов. Формализует приемку продукта, услуги или результата и подводит проект или фазу проекта к правильному завершению.   
  
Вывод: Процессы жизненного цикла программного обеспечения четко определяются с целью повышения качества получаемого продукта, улучшения коммуникаций и улучшения понимания различных аспектов программной инженерии отдельными специалистами, поддержки совершенствования процессов, поддержки управления процессами, обеспечения автоматизации процессов и т.п.

1. 4. Классификация методов верификации ПО. Место и роль статического анализа.

В данном разделе рассматриваются методы верификации ПО, в основном нацеленные на оценку технических артефактов жизненного цикла. Такие методы в имеющейся литературе разделяются на следующие группы.



Экспертиза (review) различных артефактов жизненного цикла ПО. Обычно в качестве видов экспертиз выделяют организационные экспертизы (management review), технические экспертизы (technical review), сквозной контроль (walkthrough), инспекции (inspection) и аудиты (audit). С середины 1990-х активно развиваются методы оценки архитектуры ПО на основе сценариев (scenario based software architecture evaluation), обычно не соотносимые с «традиционными» экспертизами. От других методов верификации экспертизу отличает возможность выполнять ее, используя только сами артефакты жизненного цикла, а не их модели (как в формальных методах) или результаты работы (как в динамических). Экспертиза применима к любым свойствам ПО и любым артефактам жизненного цикла и на любом этапе проекта, хотя для разных целей могут использоваться разные ее виды. Она позволяет выявлять практически любые виды ошибок, причем делать это на этапе подготовки соответствующего артефакта, тем самым минимизируя время существования дефекта и его последствия для качества производных артефактов. В то же время экспертиза не может быть автоматизирована и требует активного участия людей. Эмпирические наблюдения показывают, что эффективность экспертиз в терминах отношения количества обнаруживаемых дефектов к затрачиваемым на это ресурсам несколько выше, чем для других методов верификации. Так, различные отчеты показывают, что от 50 до 90% всех зафиксированных в жизненном цикле ПО ошибок может быть обнаружено с помощью экспертиз [56-61]. За счет их раннего обнаружения может быть достигнута существенная экономия ресурсов — затраты на обнаружения ошибки составляют от 5 до 80% от таких же затрат при использовании тестирования [59- 63]. Кроме того, регулярное участие в экспертизах является важным фактором в обучении сотрудников и способствует повышению качества результатов их работы. В то же время эффективность экспертизы существенно зависит от опыта и мотивации ее участников [58-61], организации процесса, а также от обеспечения корректного взаимодействия между различными участниками. Это накладывает дополнительные ограничения на распределение ресурсов в проекте и может приводить к конфликтам между разработчиками, если руководство проекта обращает мало внимания на коммуникативные аспекты проведения экспертиз. • Статический анализ свойств артефактов жизненного цикла ПО используется для проверки формализованных правил корректного построения этих артефактов и поиска часто встречающихся дефектов по некоторым шаблонам. Такой анализ хорошо автоматизируется и может быть практически полностью возложен на инструменты, хотя иногда необходимо вручную определить, например, принятые в проекте стандарты кодирования. Однако применим он лишь к коду или к определенным форматам представления проектных артефактов, и способен обнаруживать только ограниченный набор типов ошибок. Одной из известных проблем статического анализа является также следующая дилемма: либо используются строгие методы анализа, не допускающие пропуска ошибок (тех типов, что ищутся), но приводящие к большому количеству сообщений о возможных ошибках, которые таковыми не являются, либо точным является набор сообщений об ошибках, но возникает возможность пропустить ошибку. Инструменты автоматической верификации на основе статического анализа применяются достаточно широко, поскольку не требуют специальной подготовки и достаточно удобны в использовании. Большинство техник статической проверки корректности программ, доказавших свою эффективность на практике, рано или поздно становятся частью компиляторов или даже преобразуются в семантические правила языков программирования. • Формальные методы верификации используют для анализа свойств ПО формальные модели требований, поведения ПО и его окружения. Анализ формальных моделей выполняется с помощью специфических техник, таких как дедуктивный анализ (theorem proving), проверка моделей (model checking) или абстрактная интерпретация (abstract interpretation). Формальные методы применимы только к тем свойствам, которые выражены формально в рамках некоторой математической модели, а также к тем артефактам, для которых можно построить адекватную формальную модель. Соответственно, для использования таких методов в проекте необходимо затратить значительные усилия на построение формальных моделей. К тому же, построить такие модели и провести их анализ могут только специалисты по формальным методам, которых не так много, и чьи услуги стоят достаточно дорого. Построение формальных моделей нельзя автоматизировать, для этого всегда необходим человек. Анализ их свойств в значительной мере может быть автоматизирован, и сейчас уже есть инструменты, способные анализировать формальные модели промышленного уровня сложности, однако чтобы эффективно пользоваться ими часто тоже требуется очень специфический набор навыков и знаний (в специфических разделах математической логики и алгебры). Тем не менее, в ряде областей, где последствия ошибки в системе могут оказаться чрезвычайно дорогими, формальные методы верификации активно используются. Они способны обнаруживать сложные ошибки, практически не выявляемые с помощью экспертиз или тестирования. Кроме того, формализация требований и проектных решений возможна только при их глубоком понимании, и поэтому вынуждает провести тщательнейший анализ этих артефактов, для чего часто необходима совместная работа специалистов по формальным методам и экспертов в предметной области. В последние 10 лет появились основанные на формальных методах инструменты [64-68], решающие достаточно ограниченные задачи верификации ПО из определенного класса, но зато способные эффективно работать в промышленных проектах и требующие для применения минимальных специальных навыков и знаний. Гораздо чаще, чем к программам, формальные методы верификации на практике применяются к аппаратному обеспечению [69-72]. Их использование в этой области имеет более долгую историю, что привело к созданию более зрелых методик и инструментов. Это обусловлено более высокой стоимостью ошибок для аппаратного обеспечения, более однородной его структурой и более простыми примитивными элементами, более широким многократным использованием проектной информации, а также большей привычностью строгих ограничений и точных описаний для инженеров. • Динамические методы верификации, в рамках которых анализ и оценка свойств программной системы делаются по результатам ее реальной работы или работы некоторых ее моделей и прототипов. Примерами такого рода методов являются обычное тестирование или имитационное тестирование, мониторинг, профилирование. Для применения динамических методов необходимо иметь работающую систему или хотя бы некоторые ее компоненты, или же их прототипы, поэтому нельзя использовать их на первых стадиях разработки. Зато с их помощью можно контролировать характеристики работы системы в ее реальном окружении, которые иногда невозможно аккуратно проанализировать с помощью других подходов. Динамически методы позволяют обнаруживать в ПО только ошибки, проявляющиеся при его работе, а, например, дефекты удобства сопровождения найти не помогут, однако, обнаруживаемые ими ошибки обычно считаются более серьезными. Для применения динамических методов верификации обычно требуется дополнительная подготовка — создание тестов, разработка тестовой системы, позволяющей их выполнять или системы мониторинга, позволяющей контролировать определенные характеристики поведения проверяемой системы. Но системы тестирования, профилирования или мониторинга могут быть сделаны один раз и использоваться многократно для широких классов ПО, лишь сами тесты необходимо готовить заново для каждой проверяемой системы. В то же время подготовка тестов на ранних этапах создания ПО позволяет обнаружить множество дефектов в описании требований и проектных документах — фактически, разработчики тестов вынуждены в ходе своей деятельности выполнять экспертизу артефактов, служащих основой для тестов. Создание набора тестов, позволяющих получить адекватную оценку качества сложной системы, является довольно трудоемкой задачей. Однако среди разработчиков промышленного ПО сложилось (не вполне верное) мнение, что тестирование является наименее ресурсоемким методом верификации, поэтому на практике оно используется для оценки свойств ПО очень широко. При этом чаще всего применяются не слишком надежные, но достаточно дешевые техники, такие как (нестрогое) вероятностное тестирование, при котором тестовые данные генерируются случайным образом, или же тестирование на основе простейших сценариев использования, проверяющие лишь наиболее простые ситуации. Синтетические методы. В последние 10-15 лет появилось множество исследовательских работ и инструментов, в рамках которых применяются элементы нескольких перечисленных выше видов верификации. Так, в отдельные области выделились динамические методы, использующие элементы формальных, — тестирование на основе моделей (model-based testing, model driven testing) [73] и мониторинг формальных свойств (runtime verification, passive testing). Ряд инструментов построения тестов существенно использует как формализацию некоторых свойств ПО, так и статический анализ кода. Общая идея таких методов вполне понятна — попытаться сочетать преимущества основных подходов к верификации, купировав их недостатки. Представленная здесь классификация методов верификации скорее обусловлена историческими причинами, чем основана на существенных характеристиках самих этих методов. Исследователи и разработчики новых методов и инструментов, пытаясь соотнести свои работы с работами предшественников, обычно ищут их в рамках одной из указанных групп, поэтому в настоящий момент такая схема достаточно удобна. Однако, как уже было сказано, в последнее время создаются синтетические методы, сочетающие элементы всех остальных, и через 5-10 лет, после появления достаточно большого количества таких подходов, потребуется более детальная и содержательная классификация методов верификации.

В большинстве случаев под статическим анализом подразумевают анализ, осуществляемый с помощью автоматизированных инструментов исходного или исполняемого кода.

Исторически первые инструменты статического анализа (часто в их названии используется слово lint) применялись для нахождения простейших дефектов программы. Они использовали простой поиск по сигнатурам, то есть обнаруживали совпадения с имеющимися сигнатурами в базе проверок. Они применяются до сих пор и позволяют определять "подозрительные" конструкции в коде, которые могут вызвать падение программы при выполнении.

Недостатков у такого метода немало. Основным является то, что множество "подозрительных" конструкций в коде не всегда являются дефектами. В большинстве случаев такой код может быть синтаксически правильным и работать корректно. Соотношение "шума" к реальным дефектам может достигать 100:1 на больших проектах. Таким образом, разработчику приходится тратить много времени на его отсеивание от реальных дефектов, что отменяет плюсы автоматизированного поиска.

Несмотря на очевидные недостатки, такие простые утилиты для поиска уязвимостей до сих пор используются. Обычно они распространяются бесплатно, так как коммерческого применения они, по понятным причинам, не получили.

Второе поколение инструментов статического анализа в дополнение к простому поиску совпадений по шаблонам оснащено технологиями анализа, которые до этого применялись в компиляторах для оптимизации программ. Эти методы позволяли по анализу исходного кода составлять графы потока управления и потока данных, которые представляют собой модель выполнения программы и модель зависимостей одних переменных от других. Имея данные, графы можно моделировать, определяя, как будет выполняться программа (по какому пути и с какими данными).

Поскольку программа состоит из множества функций, процедур модулей, которые могут зависеть друг от друга, недостаточно анализировать каждый файл по отдельности. Для полноценного межпроцедурного анализа необходимы все файлы программы и зависимости.

Основным достоинством этого типа анализаторов является меньше количество "шума" за счет частичного моделирования выполнения программ и возможность обнаружения более сложных дефектов.

3 Классификация методов верификации ПО. Место и роль статического анализа.  
Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам. В частности, верификация проверяет соответствие между нормами стандартов, описанием требований (техническим заданием) к ПО, проектными решениями, исходным кодом, пользовательской документацией и функционированием самого ПО. Кроме того, проверяется, что требования, проектные решения, документация и код оформлены в соответствии с нормами и стандартами, принятыми в данной стране, отрасли и организации при разработке ПО, а также - что при их создании выполнялись все указанные в стандартах операции, в нужной последовательности. Обнаруживаемые при верификации ошибки и дефекты являются расхождениями или противоречиями между несколькими из перечисленных документов, между документами иреальной работой программы, между нормами стандартов и реальным процессами разработки и сопровождения ПО. При этом принятие решения о том, какой именно документ подлежит исправлению (может быть, и оба) является отдельной задачей.  
Методы верификации:  
Метод аксиоматической семантики Хоара  
Метод индуктивных утверждений Флойда  
Доказательное программирование (proofing programming)  
Автоматическое доказательство теорем (Theorem proving)  
Проверка моделей (Model checking)  
Символьное выполнение (Symbolic execution)  
Абстрактная интерпретация (Abstract Interpretation)  
Доказательное программирование, или Формальная верификация — формальное доказательство соответствия или несоответствия формального предмета верификации его формальному описанию. Предметом выступают алгоритмы, программы и другие доказательства. Тестирование программного обеспечения не может доказать, что система, алгоритм или программа не содержит никаких ошибок и дефектов и удовлетворяет определённому свойству. Это может сделать формальная верификация. Формальная верификация может использоваться для проверки таких систем, как программное обеспечение, представленное в виде исходных текстов, криптографические протоколы, комбинаторные логические схемы, цифровые схемы с внутренней памятью.  
Автоматическое доказательство теорем — доказательство, реализованное программно. В основе лежит аппарат математической логики. Используются идеи теории искусственного интеллекта. Процесс доказательства основывается на логике высказываний и логике предикатов.  
Проверка моделей (проверка на модели, англ. model checking) — метод автоматической формальной верификации параллельных систем с конечным числом состояний. Позволяет проверить удовлетворяет ли заданная модель системы формальным спецификациям. Обычно спецификации задаются на языке формальной логики. Для спецификации аппаратного и программного обеспечения, как правило, применяют темпоральную логику — специальный язык, позволяющий описывать поведение системы во времени.  
Стати́ческий ана́лиз ко́да (англ. static code analysis) —анализ программного обеспечения, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ. В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-нибудь вид объектного кода, например P-код или код на MSIL. Термин обычно применяют к анализу, производимому специальным ПО, тогда как ручной анализ называют "program understanding", "program comprehension" (пониманием или постижением программы).  
В зависимости от используемого инструмента глубина анализа может варьироваться от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Способы использования полученной в ходе анализа информации также различны — от выявления мест, возможно содержащих ошибки (утилиты типа lint), до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации).  
Некоторые люди считают программные метрики и обратное проектирование формами статического анализа. Получение метрик и статический анализ часто совмещаются, особенно при создании встраиваемых систем (software quality objectives).   
В последнее время статический анализ всё больше используется в верификации свойств ПО, используемого в компьютерных системах высокой надёжности, особенно критичных для жизни (Safety-critical (англ.)). Также применяется для поиска кода, потенциально содержащего уязвимости (иногда это применение называется Static Application Security Testing, SAST).   
Статический анализ постоянно применяется в следующих областях:  
ПО для медицинских устройств.   
ПО для ядерных станций и систем защиты реактора (Reactor Protection Systems).   
ПО для авиации (в комбинации с динамическим анализом)   
По данным VDC, примерно 28% разработчиков встраиваемого ПО используют средства статического анализа, и 39% собираются начать их использование в течение 2 лет.  
Типы ошибок, обнаруживаемых статическими анализаторами  
Неопределённое поведение — неинициализированные переменные, обращениек NULL-указателям. О простейших случаях сигнализируют и компиляторы.  
Нарушение блок-схемы пользования библиотекой. Например, для каждого fopen нужен fclose. И если файловая переменная теряется раньше, чем файл закрывается, анализатор может сообщить об ошибке.  
Типичные сценарии, приводящие к недокументированному поведению.  Стандартная библиотека языка Си известна большим количеством неудачных технических решений. Некоторые функции, например, gets, в принципе небезопасны, sprintf и strcpy безопасны лишь при определённых условиях.  
Переполнение буфера — когда компьютерная программа записывает данные за пределами выделенного в памяти буфера.  
  
Вывод: Статический анализ включает в себя множество методик, каждая из которых помогает разработчику в поиске разнообразных ошибок и проблем на разных стадиях процесса разработки. Он является особенно эффективным для обнаружения неопределенного поведения.  
  
  
5 Цели, задачи, возможности, ограничения статического анализа исходных кодов ПО.  
Многим трудно поверить, что были такие времена, когда можно было распечатать весь исходный код программы, прочитать его от начала до конца и увидеть ошибки или некорректную работу каких-либо функций. Сложность современных приложений делает этот подход нереалистичным для человека, но некоторые типы интеллектуального ПО вроде средств статического анализа вполне справляются с этим.  
Статический анализ программного обеспечения заключается в логическом распознавании поведения блока кода на основе содержащихся в нем инструкций. Инструмент статического анализа выполняет разбор исходного кода, постоянно пополняя свою базу знаний, используемую для проверки выражений и определения, правилен ли их вывод. При статическом анализе код на самом деле не выполняется — соответствующий инструмент просто считывает строки кода и вычисляет разные вещи. Какие именно?  
В целом, статический анализ позволяет выявлять возможные ошибки в коде, а также выяснять, насколько определенные сегменты кода отвечают ожиданиям и спецификациям.  
Со времен великой работы, проделаннойАланом Тьюрингом в 30-е годы прошлого века, известно, что автоматически определить со 100%-ной надежностью, завершится ли данная программа без ошибок, невозможно. В любом случае компьютерная наука в основном опирается на аппроксимацию, и возможность заблаговременно получить предупреждение о вероятной ошибке в каких-то строках кода очень полезна. Теперь представьте размеры современных проектов и вы получите представление о настоящей ценности. Инструмент статического анализа, возможно, не даст вам абсолютной уверенности, но, работая в реальном времени, пока вы пишете свои классы, он обеспечивает мгновенную обратную связь, указывая на то, что может быть ошибочно в вашей реализации.  
Тестирование принципиально отличается от статического анализа. Как разработчик вы активно пишете тест, но пассивно наблюдаете за статическим анализом. Если ваши тесты на самом деле не охватывают важные условия или не проверяют их с использованием важных значений, результаты не будут иметь смысла. Средство статического анализа выдает предупреждения о фактах (как их понимает это средство), которые нарушают некоторые из сконфигурированных внутренних правил. В целом, если вы не получаете от такого инструмента почти никаких предупреждений, это с высокой вероятностью указывает на хорошее качество вашего ПО. С другой стороны, получение предупреждений еще не означает, что ваше ПО «забагованное» и рухнет при первом же запуске. При статическом анализе в большей мере идентифицируются трудные в отлове ошибки, относящиеся к граничным случаям, которые потенциально способны обрушить ваше приложение. Как и тестирование, статический анализ позволяет обнаруживать дефекты на весьма ранних этапах разработки. [2]  
Цель статистического анализа: установление первопричины существующих проблем и предоставление объективной информации для принятия решений по корректирующим и предупреждающим действиям.  
Задачи, решаемые программами статического анализа кода можно разделить на 3 категории:  
1. Выявление ошибок в программах.   
2. Рекомендации по оформлению кода.Некоторые статические анализаторы позволяют проверять, соответствует ли исходный код, принятому в компании стандарту оформления кода. Имеется в виду контроль количества отступов в различных конструкциях, использование пробелов/символов табуляции и так далее.  
3. Подсчет метрик. Метрика программного обеспечения - это мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций. Существует большое количество разнообразных метрик, которые можно подсчитать, используя те ли иные инструменты.  
Есть и другие способы использования инструментов статического анализа кода. Например, статический анализ можно использовать как метод контроля и обучения новых сотрудников, еще недостаточно знакомых с правилами программирования в компании.  
Существует большое количество коммерческих и бесплатных статических анализаторов кода.[4] Список языков, для которых существуют статические анализаторы кода, также достаточно велик (Си, Си++, C#, Java, Ada, Fortran, Perl, Ruby, ...).  
Как и у любой другой методологии выявления ошибок, у статического анализа есть свои сильные и слабые стороны. Важно понимать, что нет идеального метода тестирования программ. Для разных классов программного обеспечения разные методики будут давать разные результаты. Добиться высокого качества программы можно только используя сочетание различных методик.  
Главное преимущество статического анализ состоит в возможности существенной снижении стоимости устранения дефектов в программе. Чем раньше ошибка выявлена, тем меньше стоимость ее исправления.   
Инструменты статического анализа позволяют выявить большое количество ошибок этапа конструирования, что существенно снижает стоимость разработки всего проекта. Например, статический анализатор кода PVS-Studio может запускаться в фоновом режиме сразу после компиляции и в случае нахождения потенциальной ошибки уведомит программиста. [5]   
Другие преимущества статического анализа кода:  
1. Полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получаютуправление крайне редко. Такие участки кода, как правило, не удается протестировать другими методами. Это позволяет находить дефекты в обработчиках редких ситуаций, в обработчиках ошибок или в системе логирования.  
2. Статический анализ не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа. Это позволяет находить скрытые ошибки, которые могут проявить себя только через несколько лет. Например, это ошибки неопределенного поведения. Такие ошибки могут проявить себя при смене версии компилятора или при использовании других ключей для оптимизации кода. [6]   
3. Можно легко и быстро обнаруживать опечатки и последствия использования Copy-Paste. Как правило, нахождение этих ошибок другими способами является кране неэффективной тратой времени и усилий. Обидно после часа отладки обнаружить, что ошибка заключается в выражении вида "strcmp(A, A)". Обсуждая типовые ошибки, про такие ляпы, как правило, не вспоминают. Но на практике на их выявление тратится существенное время.  
Недостатки статического анализа кода  
1. Статический анализ, как правило, слаб в диагностике утечек памяти и параллельных ошибок. Чтобы выявлять подобные ошибки, фактически необходимо виртуально выполнить часть программы. Это крайне сложно реализовать. Также подобные алгоритмы требуют очень много памяти и процессорного времени. Как правило, статические анализаторы ограничиваются диагностикой простых случаев. Более эффективным способом выявления утечек памяти и параллельных ошибок является использование инструментов динамического анализа.  
2. Программа статического анализа предупреждает о подозрительных местах. Это значит, что на самом деле код, может быть совершенно корректен. Это называется ложно-позитивными срабатываниями. Понять, указывает анализатор на ошибку или выдал ложное срабатывание, может только программист. Необходимость просматривать ложные срабатывания отнимает рабочее время и ослабляет внимание к тем участкам кода, где в действительности содержатся ошибки.  
Ошибки, обнаруживаемые статическимианализаторами весьма разнообразны. Вот, например, список диагностик, которые реализованы в инструменте PVS-Studio. Некоторые анализаторы специализируются на определенной области или типах дефектов. Другие, поддерживают определенные стандарты кодирование, например MISRA-C:1998, MISRA-C:2004, Sutter-Alexandrescu Rules, Meyers-Klaus Rules и так далее.  
Область статического анализа активно развивается, появляются новые диагностические правила и стандарты, некоторые правила устаревают. Поэтому нет смысла пытаться сравнить анализаторы, на основании списков обнаруживаемых дефектов. Единственный способ сравнить инструменты, это проверить с их помощью набор проектов и посчитать найденных ими количество настоящих ошибок. [7] [8]  
Вывод: Процесс статистического анализа определяет содержание и порядок деятельности по преобразованию данных первичных измерений параметров продукции и процессов жизненного цикла в информацию по уменьшению вариабельности.  
5 Проблема отчуждаемости критического ПО.  
В общем случае критическое ПО может быть отнесено к одному из двух типов:  
а) Автономный комплекс ПО, реализующий критические функции (standalone critical SW)  
б) ПО информационно-управляющих систем (ИУС) критического применения (SW critical I&C system)  
Риски связанные с безопасностью критического ПО всегда оцениваются на системном уровне в контексте использования ИУС, функциональность которой реализуется с использованием ПО, или автономного программного комплекса.  
Необходимым условием для проведения анализа критичности ПО и управления рисками его использования по назначению является реализация при проектировании свойства отчуждаемость. Это означает наличие технической (программной) документации, обеспечивающей объективную возможность проведения анализа и испытаний критического ПО при независимой верификации, сертификации, аттестации и сопровождении его в эксплуатации.  
К числу важнейших характеристик, определяющих качество критического ПО наряду с Функциональной и Информационной безопасностью (Safety, Security) относится характеристикаГарантоспособность (Dependability). В общем случае Гарантоспособность основывается на методах достижения бездефектности, включая предупреждение дефектов при проектировании, обнаружение, диагностирование и «извлечение дефектов» (коррекции ПО) при тестировании и парирование оставшихся скрытых дефектов при использовании критического ПО на основе реализации принципа разнообразия (многоверсионности).  
Оценка (прогноз) вероятности существования скрытых (латентных) дефектов критического ПО является исходной позицией для определения характеристик Гарантоспособность и Безопасность критического ПО. [3]  
Именно необходимость достижения упомянутых выше показателей при разработке критического ПО определяет специфические задачи и особенности технологий критической ПИ. Специфическим требованием к таким технологиям является достижение предельно-возможных значений бездефектности в результате верификации, валидации и квалификационных испытаний критического ПО. Риски ущерба из-за скрытых дефектов критического ПО могут быть неприемлемо высокими.  
К числу таких технологий, например, относятся методы Model Checking верификации (верификации моделей ПО), использующие темпоральную логику ПО и модифицированный метод полимодельной Model Checking верификации, основанный на использовании класса инварианто-ориентированных моделей критического ПО. Метод разработан в рамках проекта (гранта) УНТЦ специалистами и учеными Харьковского Национального аэрокосмического университета им. Н.Е.Жуковского ХАИ, Национального университета им.Каразина и организациями промышленности. [10]  
Вывод: Оценка вероятности существования скрытых дефектов критического ПО является исходной позицией для определения характеристик гарантоспособности и безопасности критического ПО.