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]

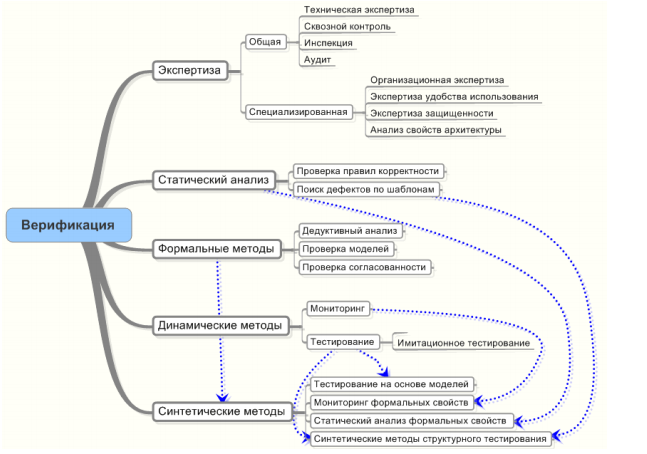
1.Идентификация, анализ и детализация объекта изучения, представленного соответствующими процессами системного и программного контекстов Опорной (ссылочной) модели процессов ЖЦ ПО   
Под понятием «программное обеспечение, критичное с точки зрения безопасности» (Safety-Critical Software – для краткости 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.

**7. 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 такая характеристика как Гарантоспособность (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 гарантоспособности и безопасности критического ПО. [1]

Именно необходимость достижения упомянутых выше показателей при разработке критического ПО определяет специфические задачи и особенности технологий. 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. [2]

Выводы по разделу:

Оценка вероятности существования скрытых дефектов критического ПО является исходной позицией для определения характеристик гарантоспособности и безопасности критического ПО.

**8. Теоретические основы анализа текстов ПО на языках высокого уровня.**

Преобразование программы, написанной на языке высокого уровня, в двоичный код, требует пристального внимания к многочисленным деталям - того, что успешно может делать компьютер под управлением транслирующей программы.

Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой.

Транслирующие программы делятся на две категории: интерпретаторы и компиляторы.  
Интерпретатор преобразует небольшой фрагмент исходной программы в машинные команды и, лишь дождавшись, когда компьютер их выполнит, переходит к обработке следующего фрагмента.

Компилятор, наоборот, транслирует всю программу, написанную на языке высокого уровня, и помещает команды в память компьютера, не выполняя их; компилированную программу можно сохранить, чтобы в дальнейшем использовать.  
Каждый из этих способов преобразования имеет свои достоинства и недостатки.

Компилированные программы выполняются быстрее, чем интерпретируемые; однажды компилированная программа не требует в дальнейшем компилятора, и компьютеру не приходится одновременно и транслировать, и выполнять программу.

Программы, написанные на языках, ориентированных на интерпретацию, требуют присутствия в памяти компьютера интерпретатора, который осуществляет трансляцию программы в ходе ее выполнения.

Благодаря построчной трансляции интерпретатор полезен как при отладке, так и при трансляции программ, подверженных частым изменениям; исправленную программу можно сразу запустить, чтобы проверить ее работу. При использовании компилятора исправленную программу приходится перекомпилировать.

Как компилятор, так и интерпретатор должны соответствовать правилам конкретного языка высокого уровня, который они транслируют. Подобно тому, как правила грамматики описывают естественный язык, правила формального языка управляют работой программиста, указывая, как можно соединять слова и символы, используемые в языке, при построении сложных выражений и задавая правила форматирования, в том числе употребления пробелов и знаков пунктуации. В транслирующих программах грамматика является основой преобразования понятий исходной программы в машинный код. [3]  
  
Шаги преобразования программ

Программа высокого уровня описывает логику решения задачи и удовлетворяет правилам записи, в соответствии с которыми интерпретатор или компилятор осуществляют трансляцию. Когда транслирующая программа (компилятор либо интерпретатор) запускается для обработки исходного кода, в память компьютера вместе с самим транслятором загружается и программа на языке высокого уровня.

Затем транслятор начинает литера за литерой читать исходный код, обрабатывая его в несколько этапов.

* Исходная программа. Программа должна описывать логику решения задачи и удовлетворять правилам записи, в соответствии с которыми интерпретатор или компилятор осуществляют трансляцию.
* Интерпретатор. Трансляция интерпретируемой исходной программы проходит в четыре этапа:

1. лексический анализ - группы литер исходного языка преобразуются в символы, понятные интерпретатору;
2. синтаксический анализ - символы размещаются в иерархическом порядке, отражающем логику программы;
3. контроль типов - проверяется согласованность символов друг с другом, что позволяет выявить ошибки в программе;
4. генератор кода посылает машинные команды в процессор, где они немедленно выполняются.

* Компилятор. Подобно интерпретатору, компилятор также пропускает программу через этапы лексического анализа, синтаксического анализа, контроля типов и генерации кода. Но в этом случае оттранслированная программа не выполняется немедленно, а сохраняется в памяти компьютера; дополнительный этап в работе компилятора - так называемая оптимизация - улучшает результат трансляции, повышая эффективность полученной программы.
* Целевой код. На выходе компилятора можно получить либо последовательность машинных команд низкого уровня, называемую объектным кодом, либо последовательность команд языка ассемблера. Если целевой код является языком ассемблера, то перед выполнением его следует оттранслировать ассемблером.

На каждом этапе информация, полученная от предыдущего этапа, обрабатывается и передается дальше. Если какое-либо правило языка оказывается нарушенным, запускается подпрограмма обработки ошибок, которая сообщает программисту о найденной ошибке.[4]

Выводы по разделу:

Как компилятор, так и интерпретатор должны соответствовать правилам конкретного языка высокого уровня, который они транслируют.

**9. Синтаксический разбор текста ПО. Парсинг. Формальный анализ  исходных текстов ПО.**

Синтаксический анализатор (синтаксический разбор) — это часть компилятора, которая отвечает за выявление основных синтаксических конструкций входного языка. В задачу синтаксического анализа входит:

* Поиск и выделение синтаксических конструкций в тексте исходной программы;
* Установка типа и проверка правильности каждой синтаксической кострукции;
* Представление синтаксических конструкций в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализ, парсинг - это процесс сопоставления линейной последовательности лексем (слов, фраз) языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

### В основе синтаксического анализатора лежит распознаватель текста входной программы на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик (контекстно-свободных), реже встречаются языки, которые, могут быть описаны с помощью регулярных грамматик. Чаще всего регулярные грамматики применимы к языкам ассемблера, а языки высокого уровня построены па основе синтаксиса КС-языков.

### Распознаватель дает ответ на вопрос о том, принадлежит или нет цепочка входных символов заданному языку. Однако, как и в случае лексического анализа, задача синтаксического разбора не ограничишься только проверкой принадлежности цепочки заданному языку. Необходимо выполнить все перечисленные выше задачи, которые должен решить синтаксический анализатор. В таком варианте анализатор уже не является разновидностью МП-автомата — его функции можно трактовать шире.

### Синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции не только информацию о найденных и разобранных синтаксических структурах. В таком случае он уже является преобразователем с магазинной памятью — МП-преобразователем. Синтаксический разбор — это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора бессмысленна, в то время как лексический разбор в принципе является необязательной фазой. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Сканер только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем исходной программы.

Выходом лексического анализатора является таблица лексем (или цепочка лексем). Эта таблица образует вход синтаксического анализатора, который исследует только один компонент каждой лексемы — ее тип. Остальная информация о лексемах используется на более поздних фазах компиляции при семантическом анализе, подготовке к генерации и генерации кода результирующей программы.

Синтаксический анализ (или разбор) — это процесс, в котором исследуется таблица лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным и определении синтаксиса языка.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами. Примерами конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором.

На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору. Обычно это определяет разработчик компилятора исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка. Основу любого синтаксического анализатора всегда составляет распознаватель, построенный на основе какого-либо класса КС-грамматик. Поэтому главную роль о том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

Как правило, результатом синтаксического анализа является синтаксическая структура предложения, представленная либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторой комбинации первого и второго способов представления.

Всё что угодно, имеющее "синтаксис", поддается автоматическому анализу:

* языки программирования - разбор исходного кода языков программирования, в процессе трансляции (компиляции или интерпретации);
* структурированные данные - данные, языки их описания, оформления и т. д., например, XML, HTML, CSS, ini-файлы, специализированные конфигурационные файлы и т. п.;
* SQL-запросы (DSL-язык);
* математические выражения;
* регулярные выражения (которые, в свою очередь, могут использоваться для автоматизации лексического анализа);
* формальные грамматики;
* лингвистика - человеческие языки, например, машинный перевод и другие генераторы текстов; [5]

Задачи, решаемые синтаксическим анализатором

Синтаксический анализатор должен распознать структуру предложения, а именно синтаксические зависимости слов. В результате должно быть либо построено синтаксическое дерево, либо выявлены составляющие. Обычно грамматика строится так, чтобы на выходе получалось синтаксическое дерево, позволяющее выполнять разнообразные трансформации лексического содержания с пересогласованием зависимых слов, а также легко выделять семантику, в частности - применять алгоритм взвешивания альтернативных вариантов построения дерева.

Предложение может допускать несколько альтернативных вариантов связывания слов. В этом случае анализатор пытается применять некоторые эвристики и базу знаний, но может в конце концов вернуть несколько вариантов синтаксического дерева. К такому поведению особенно склонен восходящий парсер, так как он сильно ограничен в средствах по пресечению экспоненциального роста числа вариантов. Наличие нескольких итоговых вариантов разбора может означать не только недостаточный набор правил в языковой модели, но и присущую данному предложению неоднозначность, устраняемую только учетом более широкого контекста.  
Анализируемые предложения могут иметь разную сложность, включать неизвестные слова или отступления от нормативного синтаксиса. Чтобы эффективно справляться с разными задачами, синтаксический анализатор применяет несколько разных алгоритмов, включая структурный нисходящий анализ и восходящий анализ, а также применяет семантический анализ для уточнения результатов в случае неоднозначностей.

Структурный нисходящий анализ - самый точный. Восходящий анализ работает очень быстро и способен разобрать даже очень длинные предложения, игнорируя непонятные фрагменты. [6]

Восходящий и нисходящий вероятностный парсинг

В парсере реализованы два алгоритма недетерминированного синтаксического разбора предложения - нисходящий (top-down) и восходящий (bottom-up).  
Как следует из названий, они по-разному организуют процесс распознавания синтаксических конструкций.

Нисходящий синтаксический анализ, или анализ через синтез, начинает с выдвижения предположений о крупномасштабной структуре предложения, а затем уточняет и детализирует это предположение, рекурсивно опускаясь на уровень конкретных слов. Другими словами, этот алгоритм инициирует разбор с начального нетерминала S.

Восходящий синтаксический анализ начинает разбор с конкретных слов, связывая сначала пары слов, затем подсоединяет к этим парам новые слова или другие связанные пары. Постепенно процесс связывания доходит до начального нетерминала S - то есть все слова в предложения оказываются связаны в единую структуру.

Обоим алгоритмам присущи важные общие черты:

* Они недетерминированы, то есть могут выдавать множество альтернативных вариантов синтаксического разбора, хотя оба стремятся отсечь недостоверные варианты с помощью общей модели языка. Они управляются правилами, которые задаются вручную.
* Они позволяют задавать типы связей в создаваемом синтаксическом дереве.
* Оба алгоритма реализуют идею динамического программирования для эффективного снижения времени разбора.

Выбор между восходящим и нисходящим алгоритмом обычно происходит автоматически. Встроенный в грамматический движок планировщик анализа сначала пытается распарсить предложение самым точным нисходящим алгоритмом, а в случае неудачи переключается на более устойчивый к ошибкам восходящий анализ.

Правила, управляющие синтаксическим разбором

Восходящий и нисходящий алгоритмы воплощают "классический" подход к созданию грамматики естественного языка. Этот подход заключается в создании набора правил, которые работают со словами и грамматическими категориями слов. Правила обычно пишутся на специальном предметно-ориентированном языке (DSL), чтобы упростить изменение грамматики без перекомпиляции кода парсера и дать в руки разработчику грамматики эффективный набор средств, обычно имеющих мало общего с обычными императивными языками программирования.

Грамматический движок предоставляет два алгоритмически различных подхода к построению анализатора: итерационный алгоритм на основе правил переписывания и очень мощный структурный нисходящий парсер. Несмотря на некоторое различие в математической основе, существует способ применять в обоих типах анализа общий набор правил, таким образом реализуя сложный процесс синтаксического анализа с динамическим переключением направления.

В терминах теории автоматов, набор правил реализует распознающую грамматику для предложений на естественном языке. Результатом работы анализатора является не только подтверждение того, что входное предложение с определенной достоверностью принадлежит к распознаваемому набору синтаксических конструкций и идеоматических оборотов, но и набор альтернативных синтаксических деревьев с весами в случае неоднозначности анализа.

Достоинства и недостатки структурного нисходящего парсера

Достоинства СНП:

* СНП даст 100% верный разбор предложения на составляющие, если используемая грамматика адекватно описывает такие предложения. Процесс разбора структурным парсером очень похож на математически строгое доказательство;
* СНП найдет все возможные варианты разбора предложения, если грамматика содержит неоднозначности;
* СНП способен делать ослабляющие допущения и оценивать выполненный анализ с учетом таких допущений;
* СНП может учитывать контекст любой ширины, вплоть до всего предложения. Другими словами, учитываются не только непосредственные соседи, но также связанные синтаксически части предложения, находящиеся на любом удалении от слова.

Как недостатком, так и достоинством является то, что СНП не имеет внутреннего состояния и выполняет разбор за 1 заход. Нет естественного способа остановить выполнения разбора в промежуточной стадии и получить частичные результаты. Эта особенность имеет также важное положительное следствие. Отсутствие состояния позволяет распараллеливать выполнение анализа.

Недостатки СНП:

* СНП естественным образом реализует поиск в глубину. Это может приводить к тому, что из-за отсутствия хороших эвристик очень много времени будет уделено проверке заведомо проигрышным вариантам анализа, а выигрышный путь в пространстве состояний так и не будет найден до исчерпания лимита времени;
* Следствием отсутствия состояния является трудность отладки. Отладчик парсера предоставляет некоторые ограниченные возможности, включая задание точек останова в некоторых местах правил и просмотр локального состояния вывода. Однако наличие множества параллельных процессов вывода делает интерактивную отладку крайне трудной в реализации и использовании. В реальности самым удобным способом отладки является просмотр пошаговой трассировки для полученных результатов;
* Если структура предложения не выводится правилами грамматики в точном виде, то СНП вообще ничего не выведет, если не принимать специальные меры. Под такими специальными мерами подразумевается неполный анализ;
* СНП реализует в большей степени математическую концепцию, чем языковую. В сочетании с декларативным и функциональным стилем задания правил вывода этоможет поднять порог вхождения для незнакомого с математической лингвистикой разработчика, привыкшего к императивному стилу программирования;
* СНП работает с предложением в ленивом стиле. Начав с левой границы, он постепенно выводит синтаксическую структуру, запрашивая у лексера новые и новые слова. Более того, СНП очень хорошо элегантно работают с графом токенизации, который получается в ходе лексического анализа. [7]

Выводы по разделу:

Парсинг используется вместе с лексическим анализом. При парсинге из исходного текста формируется структура данных, которое отражает синтаксическую структуру и удобно для дальнейшей обработки. Обычно, результатом синтаксического анализа будет синтаксическая структура предложения, которая представлена либо в форме дерева зависимостей, либо в форме дерева составляющих, или в форме в виде некоторой комбинации первого и второго методов представления.

Существует восходящий и нисходящий парсер, выбор между которым осуществляется автоматически.

**10. Перспективы использования статического анализа текстов ПО.**

Основной проблемой использования статического анализа текстов ПО является растущая сложность создания и апробации новых техник верификации. Все необходимое для их работы окружение — инструменты анализа исходного кода, описания формальных моделей, библиотеки для работы с внутренним представлением моделей и кода, инструменты, реализующие различные виды анализа кода и моделей, средства получения отчетов — невозможно разработать заново. Исследователю для проверки работоспособности его идеи приходится на скорую руку собирать это окружение из разнородных компонентов и библиотек, которые можно найти в свободном доступе. В лучшем случае удается создать прототип, который способен справиться с парой нужных примеров. Но таким способом невозможно создать среду, в рамках которой можно было бы проанализировать работоспособность и эффективность новой идеи в широком множестве разнообразных ситуаций, на разных видах приложений и требований к ним. Поэтому большинство новых идей применяются лишь в «тепличных условиях», а эффекты от их применения в широком контексте остаются неясными и непредсказуемыми.

Решением для упомянутых проблем могла бы стать унифицированная расширяемая среда верификации программных систем, предоставляющая общее окружение для решения задач верификации и библиотеки готовых компонентов, реализующих типовые техники. Такая среда могла бы существенно упростить интеграцию модулей, реализующих различные техники верификации, за счет унифицированных интерфейсов ее расширения.

Исследователи могли бы использовать ее для значительного снижения затрат на апробацию новых методов и анализ их работоспособности в разнообразных ситуациях. Промышленные разработчики — для интеграции нужного им набора техник в рамках единого инструмента и эффективного внедрения их в практическое использование.

Подтверждением работоспособности и эффективности интеграции различных методов верификации ПО в разнообразных ситуациях являются многочисленные синтетические методы верификации.[8]

Выводы по разделу:

Решением проблемы возрастания растущих сложностей создания новых техник верификации является  унифицированная расширяемая среда верификации программных систем, предоставляющая общее окружение для решения задач верификации и библиотеки готовых компонентов, реализующих типовые техники. Такая среда могла бы существенно упростить интеграцию модулей, реализующих различные техники верификации, за счет унифицированных интерфейсов ее расширения.

**11. Инструментальные средства и среды. Возможности Ограничения.**

Инструменты статического анализа предназначены для выявления дефектов в исходном коде программ. Принцип их работы основан на статическом анализе кода.  
Существует огромное количество инструментов статического анализа, созданных для различных языков программирования, приведем некоторые из них:

* Berkeley Lazy Abstraction Software Verification Tool (BLAST) — программа проверки моделей для языка Си.

Задача, решаемая инструментом BLAST — это проверка того, что программа удовлетворяет поведенческим требованиям к ней. BLAST реализует подход абстракция и уточнение по контрпримерам для конструирования абстрактной модели, которая затем проверяется на свойства безопасности. Абстракция строится по ходу анализа и только до требуемой точности, устанавливаемой в ходе анализа.

* FindBugs — статический анализатор кода, написанный Уильямом Пью.

Программа использует статический анализ, чтобы найти потенциальные ошибки сотни различных типов вJava коде. FindBugs работает с Java байткодом, а не с исходным кодом. Приложение распространяется и как отдельное десктопное приложение и как плагин к Eclipse, Netbeans, IntelliJ IDEA и Hudson.

* MSR Tools — генератор статистики для кода находящегося в системе контроля версий.

Позволяет выполнять расчёт различных метрик (количество строк кода, плотность ошибок и др.) для ограниченных множеств кода. Такие множества могут быть заданы различными условиями: принадлежность конкретному разработчику, нахождение в определённом файле, существование в определённой ревизии и т.п.

* ReSharper (R#) — плагин, разработанный компанией JetBrains для повышения продуктивности работы в Microsoft Visual Studio.

Проводит статический анализ кода в масштабе всего решения, предусматривает дополнительные средства автозаполнения, навигации, поиска, подсветки синтаксиса, форматирования, оптимизации и генерации кода, предоставляет 40 автоматизированных рефакторингов, упрощает юнит-тестирование в средах MSTest иNUnit и др. Поддерживает языки программирования C# и VB.NET, а также предоставляет средства для работы с ASP.NET,ASP.NET MVC, XML, XAML, сценариями сборки NAnt и MSBuild.

* SourceAnalyzer (кратко SA) — статический анализатор кода, предназначенный для поиска и анализа зависимости функций (графа вызовов).

Анализатор работает с препроцессированным кодом и представляет собой набор инструментов (бинарных файлов). Разработан графический интерфейс (QSAGUI) для визуализации графа вызовов. SA может быть полезен архитекторам ПО, программистам и тестировщикам. Основной отличительной особенностью данного статического анализатора кода является возможность быстро обработать большие объемы исходного кода (>1GB). Благодаря принципу, аналогичному раздельной компиляции, SourceAnalyzer требует не полного перестроения графа вызовов, а лишь части, в которой произошли изменения. SourceAnalyzer, являясь набором консольных инструментов, может быть легко встроен в любую автоматизированную систему;

* T-SQL Analyzer - инструмент, который может просматривать программные модули в базах данных под управлением Microsoft SQL Server 2005 или 2008 и обнаруживать потенциальные проблемы, связанные с низким качеством кода. [9]

Выводы по разделу:

Инструменты статического анализа становятся неотъемлемой частью процесса разработки программного обеспечения. Статический анализ является важным дополнением к обычно применяемым инструментам контроля качества ПО, таким как тестирование и просмотр кода. Статический анализ позволяет обнаруживать ошибки, которые с трудом выявляются на этапе обычного тестирования, в силу ограниченности набора тестов.

**12. Формальные методы анализа и верификации критического ПО на основе статического анализа.**

Для того, чтобы проверить выполнение тех или иных свойств с помощью формальных методов, необходимо формализовать свойства и проверяемый артефакт, т.е. построить формальные модели для того и другого. Модель проверяемых свойств принято называть спецификацией, а модель проверяемого артефакта — реализацией. Здесь спецификация и реализация — термины, обозначающие формальные модели, а не описание требований и реализующий их набор программ, как обычно. После этого нужно проверить некоторое формально определенное соответствие или отношение этих моделей, которое моделирует выполнение данных свойств для данного артефакта.

Поскольку и реализация, и спецификация могут быть моделями двух основных видов, логико-алгебраическими или исполнимыми, возможно 4 разных комбинации этих видов моделей. Однако на практике никогда не используют для спецификации исполнимую модель в сочетании с логико-алгебраической для реализации, поскольку в такой комбинации невозможно обеспечить большую абстрактность спецификации по сравнению с реализацией. Таким образом, остаются три разных случая.

1. И спецификация S, и реализация I представлены как логико-алгебраические модели.

В этом случае выполнение специфицированных свойств в реализации моделируется отношением выводимости, что обычно записывается как I ? S. Чаще всего для его проверки используется метод дедуктивного анализа (theorem proving), т.е. проверки того, что набор утверждений, представляющий спецификацию, формально выводится из реализации и, быть может, каких-то гипотез о поведении окружения системы, сформулированных в том же формализме, что и реализация.

1. Спецификация S является логико-алгебраической моделью, а реализация I — исполнимой.

Выполнение специфицированных свойств в реализации в этой ситуации называется отношением выполнимости и записывается как I ? S. Для его проверки используется метод проверки моделей (model checking), в рамках которого чаще всего выполнимость проверяется непосредственным исследованием всей реализации, или такой ее части, свойства которой полностью определяют свойства всей реализации в целом. Обычно эту работу выполняет не человек, а специализированный инструмент.

1. И спецификация S, и реализация I представлены как исполнимые модели.

В этом случае общепринятого названия или обозначения для выполнения специфицированных свойств в реализации нет — используются термины «симуляция» или «моделирование» (simulation), «сводимость» (reduction), «соответствие» или «согласованность» (conformance). Далее в этой работе используется последний термин — согласованность.

Для методов проверки согласованности тоже пока нет общепринятого названия. В тех случаях, когда используются модели промежуточного типа, применяемый метод определяется теми составляющими модели, которые для него наиболее существенны. Так, логики Хоара и динамические логики чаще всего используют для дедуктивного анализа, а программные контракты могут применяться в различных методах.[8]

Статический анализ предполагает построение некоторых моделей кода проверяемой системы, чаще всего, в виде размеченных графов потоков управления и данных, и анализ свойств этих моделей, например, поиск ошибок определенного рода по соответствующим им шаблонам в потоках данных. Сейчас все чаще используются специфические виды статического анализа, в рамках которых находят применения формальные модели и специализированные инструменты разрешения ограничений.

* Расширенный статический анализ (extended static checking) проверяет соответствие кода ПО требованиям, обычно записываемым тоже в коде в виде комментариев к его отдельным элементам (процедурам, типам данных и методам классов).

При этом на основе результатов анализа кода автоматически строятся формальные модели его поведения, выполнение требований для которых проверяется чаще всего с помощью дедуктивного анализа и специализированных решателей (solvers).

* Статический анализ на базе автоматической абстракции.

В рамках такого подхода на основе результатов статического анализа кода автоматически строятся более абстрактные, а потому более простые модели работы проверяемого ПО, которые затем подвергаются проверке на выполнение определенных свойств с помощью инструментов проверки моделей или решателей. Обычно проверяемые свойства фиксированы для данного инструмента или формулируются в конфигурационном файле. При нарушении требования в модели инструменты этого типа пытаются построить соответствующий сценарий работы кода. Если это не получается из-за упрощений, сделанных при построении модели, определяются элементы кода, препятствующие выполнению такого сценария, и в модель вносятся уточнения, более аккуратно описывающие работу именно этих элементов, после уточненная модель снова проверяется на выполнение заданного свойства. В итоге инструмент либо подтверждает выполнение требований, либо находит контрпример, либо завершает работу по истечении некоторого времени или из-за исчерпания ресурсов, не приходя к определенным выводам.

* Синтетическое структурное тестирование при котором после первого случайно выбранного теста остальные тесты генерируются автоматически так, чтобы обеспечить покрытие еще не покрытых ранее элементов кода.

Для выбора подходящих тестовых данных используются решатели, учитывающие символическую информацию о ранее выполненных тестах (ограничения на данные, отделяющие прошедшие тесты от еще не покрытого кода), а для построения нужных последовательностей воздействий — случайная генерация, направляемая как этой же символической информацией, так и некоторыми эвристическими абстракциями, уменьшающими пространство состояний проверяемой системы. Тестирование на основе моделей (model based testing)  сочетает разработку формальных моделей требований к проверяемому ПО и построение тестов на базе этих моделей. Структура модели при этом служит основой для критерия полноты тестирования, а ограничения модели на корректные результаты работы ПО используются в качестве тестовых оракулов, оценивающих правильность поведения ПО в ходе тестирования.

В рамках последних двух подходов (или отдельно от них) применяются специфические техники построения тестов, сами по себе сочетающие разные методы верификации.

1. Построение тестов с помощью разрешения ограничений.

Часто при разработке тестов на основе критериев полноты тестирования формулируются так называемые цели тестирования (test objectives), представляющие собой специфические ситуации, в которых необходимо проверить поведение тестируемой системы для достижения необходимой уверенности в ее корректной работе. Цель тестирования формулируется как набор ограничений на проходимые во время теста состояния системы и данные выполняемых воздействий. Для построения теста, достигающего такую цель, можно использовать специализированные решатели (solvers). Такой решатель либо автоматически находит необходимые данные и последовательность вызовов операций как решение заданной системы ограничений, либо показывает, что эта система неразрешима, т.е. заданная цель тестирования недостижима и строить нацеленные на нее тесты не имеет смысла.

1. Построение тестов как контрпримеров с помощью инструментов проверки моделей.

Другой способ построения тестов — сформулировать отрицание ограничений, задающих цель тестирования, как свойство, которое можно проверить или опровергнуть с помощью инструмента проверки моделей. Если это свойство подтверждается, значит, цель тестирования недостижима, если же оно опровергается, то инструмент строит контрпример, являющийся в данном случае необходимым тестом.  
Мониторинг формальных свойств (runtime verification, passive testing) тоже использует формальные модели требований для оценки правильности поведения проверяемой системы, но только в ходе ее обычной работы, без использования специально построенных тестов.

Как видно, все синтетические методы так или иначе пытаются соединить достоинства различных подходов к верификации, купируя их недостатки. В настоящее время достигнуты значительные успехи в разработке таких методов и внедрении их в практику промышленной разработки ПО, что подтверждает эффективность интеграции различных верификационных методов на практике. Тем не менее, несмотря на достигнутые успехи, каждый из имеющихся синтетических подходов использует лишь часть имеющегося потенциала и не предоставляет единой среды интеграции для всего многообразия различных техник верификации ПО.[10]

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 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.

Гораздо чаще, чем к программам, формальные методы верификации на практике применяются к аппаратному обеспечению. Их использование в этой области имеет более долгую историю, что привело к созданию более зрелых методик и инструментов. [9]

Выводы по разделу:

Формальные методы верификации используют для анализа свойств ПО формальных моделей требований, поведения ПО и его окружения. 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.

Существуют такие методы анализа и верификации критического ПО на основе статического анализа: расширенный статический анализ, статический анализ на базе автоматической абстракции, построение тестов с помощью разрешения ограничений, построение тестов как контрпримеров с помощью инструментов проверки моделей, мониторинг формальных свойств.

**13. Полимодельная Model-checking верификация с использованием инварианто-ориентированных моделей критического ПО. На основе статического анализа исходных текстов ПО.**

Необходимость достижения показателей, как гарантоспособность и безопасность, при разработке критического ПО определяет специфические задачи и особенности технологий критической ПО. Специфическим требованием к таким технологиям является достижение предельно-возможных значений бездефектности в результате верификации, валидации и квалификационных испытаний критического ПО. В силу того, что риски ущерба из-за скрытых дефектов критического ПО могут быть неприемлемо высокими, его необходимо тестировать. Одной из технологий проверки ПО является метод 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. [11]

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. [12]

Одной из главных проблем метода верификации моделей является огромное число состояний моделей реальных систем. Реальные системы обычно параллельны, а число состояний моделей параллельной системы растет экспоненциально с ростом числа компонентов. Возникающая при этом проблема называется проблемой «комбинаторного взрыва» или взрыва числа состояний (state explosion problem). Феномен «комбинаторного взрыва» в пространстве состояний представляет особенно серьезную проблему для верификации программ. Именно поэтому проверка на модели применялась для верификации программного обеспечения значительно реже, чем для верификации аппаратуры.  
  
Возможность применения метода model-checking к разрабатываемым реальным системам, является следствием того, что были созданы методы эффективного представления данных (так называемые символьные методы) и соответствующие алгоритмы верификации, позволяющие в значительной степени решить проблему взрыва числа состояний и выполнять методом model checking верификацию систем с громадным числом состояний – 10100 и более.

Этого удалось добиться, благодаря использованию двоичных разрешающих диаграмм – структур данных для представления булевых функций.

Новые структуры данных позволили строить компактные представления систем переходов и осуществлять их быстрые преобразования. Эта методика оказалась полезной для синхронных схем. Для асинхронных протоколов сокращение размера пространства состояний оказалось возможным за счет применения метода редукции частичных порядков. В основу этого метода положены следующие допущения: вычисления, отличающиеся порядком выполнения независимых действий, как правило, не различаются спецификациями и могут считаться эквивалентными. Поэтому нужно всего лишь проверить сокращенное пространство состояний, в котором содержится хотя бы по одному вычислению, представляющему каждый из классов эквивалентности. [16]

В качестве модели обычно используется так называемая модель Крипке. Модель Крипке (англ. Kripke structure) — это один из вариантов недетерминированного конечного автомата, который был предложен Солом Крипке. Этот вид НКА применяется при проверке моделeй для представления поведения системы.

Модель Крипке является простой абстрактной машиной, позволяющей описать идеи вычислительной машины без добавления особых сложностей. Модель представляется ориентированным графом, вершины которого описывают достижимые состояния системы, а ребра — переходы из состояния в состояние.

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.  
Важным вопросом спецификации является полнота. Метод проверки на модели позволяет убедиться, что модель проектируемой системы соответствует заданной спецификации, однако определить, охватывает ли заданная спецификация все свойства, которым должна удовлетворять система, невозможно.

Основная трудность, которую приходится преодолевать в ходе проверки на модели, связана с эффектом комбинаторного взрыва в пространстве состояний. Эта проблема возникает в системах, состоящих из многих компонентов, взаимодействующих друг с другом, а также в системах, обладающих структурами данных, способных принимать большое число значений. [11]

Верификация на модели может проводиться на самых ранних этапах проектирования по заранее разработанной модели, что существенно экономит время и затраты на исправление ошибок. Существующие методы верификации на модели достаточно эффективны, они поддержаны системами автоматического выполнения. Однако у верификации на модели есть ряд недостатков:

1. Проверяется не реальная система, а ее абстрактная модель (это один из главных недостатков верификации на модели).

Такая модель может быть неадекватной, не сохранять существенных черт исходной системы, и в этом случае проверка спецификации для модели не выявит ошибок, имеющихся в реальной системе. С другой стороны, в модели могут появиться свойства, которых в реальной системе не существует. Таким образом, ценность результатов, полученных при верификации на модели, напрямую зависит от адекватности модели и, следовательно, от уровня профессионализма человека, строящего модель;

1. Язык спецификации свойств системы может быть неполным, недостаточным для формулировки всех желаемых требований к поведению системы.

Выполнение любого множества проверок системы в этом случае не гарантирует ее правильности;

1. Верификация на модели (так же, как и тестирование) не может гарантировать абсолютной правильности системы, поскольку невозможно формально определить, что означает "полное отсутствие ошибок";
2. Сама система верификации может содержать ошибки;
3. Выполнение верификации на модели требует достаточно высокой квалификации персонала (несмотря на то, что существенная часть процесса верификации на модели обычно автоматизирована).

Построение адекватных моделей реальных систем является весьма непростой задачей. Иногда модель, для которой проводится верификация, допускает такие траектории вычисления, которые не могут произойти в реальной системе. Исключение таких траекторий из процесса анализа системы требует глубокого понимания процесса абстрагирования. Формулировка требований на языке спецификаций требует знаний логики (обычно это темпоральная логика), свойств параллельных систем и способов выражения этих свойств на языке логики. инварианто-ориентированный подход.

Для преодоления установленных недостатков при верификации на моделях предложен инварианто-ориентированный подход. Он основан на использовании инвариантов для специфицирования атрибутов и состояний, которые может принимать ПО при реализации всех маршрутов обработки данных. [12]

Выводы по разделу:

Для выявления скрытых дефектов критического ПО применяют методы Model Checking верификации, использующие темпоральную логику ПО и модифицированный метод полимодельной Model Checking верификации.

1 - ГОСТ\_Р\_12207.pdf

2 - <http://software.intel.com/ru-ru/articles/Static-code-analysis-ru>

3 - С. В. Зеленов, С. А. Зеленова, А. С. Косачев, А. К. Петренко. Генерация тестов для компиляторов и других текстовых процессоров. Программирование, 29(2):59–69, 2003.

4 - <http://chernykh.net/content/view/221/234/>

5 – Молчанов А.Ю. Системное программное обучение: учебник для вузов – СПб/Питер, 2006. – 396с.

6 - А. Ахо, Р. Сети, Дж. Ульман "Компиляторы: принципы, технологии и инструменты",М.: "Вильямс", 2001, 768 стр.

7 - Шулянский Д.В. Задача синтаксического анализа текста для тестирования языковых знаний

8 – Кулямин В. В. Перспективы интеграции методов верификации программного обеспечения

9 - <https://ru.wikipedia.org>

10 - Кулямин В.В. Методы верификации программного обеспечения — Москва. Изд-во БГУЭП, 2004. — 84 с.

11 - Кулямин В.В. Тестирование на основе моделей

12 - Кларк Э.М. Верификация моделей программ: Model Checking / Кларк Э.М., Грамберг О., Пелед Д.: пер. с англ. под ред. Р. Смелянского. – М.: МЦНМО, 2002. – 416 с.