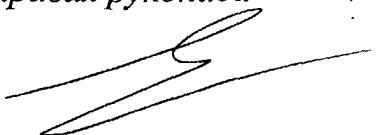


Федеральное государственное бюджетное образовательное учреждение  
высшего профессионального образования  
**«Петербургский государственный университет путей сообщения»**  
(ФГБОУ ВПО ПГУПС)

*На правах рукописи*



04201352747

ДИАСАМИДЗЕ СВЕТЛАНА ВЛАДИМИРОВНА

**МЕТОД ВЫЯВЛЕНИЯ НЕДЕКЛАРИРОВАННЫХ  
ВОЗМОЖНОСТЕЙ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ  
СТРУКТУРИРОВАННЫХ МЕТРИК СЛОЖНОСТИ**

05.13.19 – МЕТОДЫ И СИСТЕМЫ ЗАЩИТЫ ИНФОРМАЦИИ,  
ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ

Диссертация на соискание ученой степени кандидата технических наук

Научный руководитель:  
доктор технических наук, профессор  
Еремеев М.А.

Санкт-Петербург – 2012

## ОГЛАВЛЕНИЕ

<b>ВВЕДЕНИЕ .....</b>	<b>4</b>
<b>1 СИСТЕМНЫЙ АНАЛИЗ ПРОБЛЕМЫ ПОДТВЕРЖДЕНИЯ СООТВЕТСТВИЯ ПРОГРАММНЫХ СРЕДСТВ ПО ТРЕБОВАНИЯМ БЕЗОПАСНОСТИ ИНФОРМАЦИИ НА ЖЕЛЕЗНОДОРОЖНОМ ТРАНСПОРТЕ .....</b>	<b>11</b>
1.1 Тенденции развития системы подтверждения соответствия программных средств по требованиям качества и безопасности на железнодорожном транспорте.....	11
1.2 Анализ стандартов критериального аппарата и проблемных вопросов подтверждения соответствия программных средств по требованиям безопасности информации.....	21
1.3 Общая характеристика методов исследования программ, верификации и выявления недекларированных возможностей программных средств	29
1.4 Анализ возможностей использования метрик сложности для задач формальной верификации и выявления недекларированных возможностей программ .....	50
1.5 Постановка научной задачи исследования .....	59
<b>2 ОБОСНОВАНИЕ ПОДХОДА К СТРУКТУРИРОВАНИЮ МОДЕЛЕЙ ПРЕДСТАВЛЕНИЯ ПРОГРАММ И МЕТРИК СЛОЖНОСТИ НА ОСНОВЕ АНАЛИЗА СЕМАНТИК.....</b>	<b>61</b>
2.1 Анализ трактовок и подходов к формальному описанию семантики языков программирования.....	61
2.2 Выбор моделей представления программ на основе структурированных семантик .....	67
2.3 Построение логической модели программы с использованием граф- ориентированных схем Янова .....	80
2.4 Построение метрической модели на основе требований к контролю отсутствия недекларированных возможностей.....	85

Выводы по главе 2 .....	88
<b>3 МЕТОД ВЫЯВЛЕНИЯ НЕДЕКЛАРИРОВАННЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ СТРУКТУРИРОВАННЫХ МЕТРИК СЛОЖНОСТИ .....</b>	<b>90</b>
3.1 Общая характеристика метода выявления недекларированных возможностей программ на основе структурированных метрик сложности .....	90
3.2 Построение метрического базиса как совокупности структурированных метрик топологической и информационной сложности .....	91
3.3 Приведение программы к продуктивной структуре на основе канонического представления схем Янова .....	115
3.4 Формирование системы критериев для классификации выявленных дефектов как подобных НДВ .....	128
Выводы по главе 3 .....	135
<b>4 МЕТОДИКА ВЫЯВЛЕНИЯ НЕДЕКЛАРИРОВАННЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММ И ПРАКТИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ СТРУКТУРИРОВАННЫХ МЕТРИК СЛОЖНОСТИ .....</b>	<b>138</b>
4.1 Место и роль метрических оценок в системе статического анализа программных средств.....	138
4.2 Методика выявления недекларированных возможностей программ с использованием структурированных метрик топологической и информационной сложности.....	144
Выводы по главе 4 .....	149
<b>ЗАКЛЮЧЕНИЕ.....</b>	<b>151</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....</b>	<b>153</b>

## ВВЕДЕНИЕ

Угрозы информационной безопасности, как указано в Стратегии национальной безопасности Российской Федерации до 2020 года, предотвращаются за счет совершенствования безопасности функционирования информационных и телекоммуникационных систем критически важных объектов инфраструктуры и объектов повышенной опасности в Российской Федерации, повышения уровня защищенности корпоративных и индивидуальных информационных систем. Также, на основании документа ФСТЭК России от 18.05.2007 г. «Общие требования по обеспечению безопасности информации в ключевых системах информационной инфраструктуры» система управления железнодорожным транспортом определяется как ключевая система информационной инфраструктуры. Это, в свою очередь, определяет информационно-управляющие, автоматизированные системы, программные и программно-технические комплексы инфраструктуры железнодорожного транспорта и железнодорожного подвижного состава, в том числе и высокоскоростного, как системы управления критически важными объектами или технологическими процессами. Таким образом, для данных систем на железнодорожном транспорте выдвигаются усиленные требования к обеспечению информационной безопасности и защите информации.

Важную роль при обеспечении функциональной и информационной безопасности информационно-управляющих и автоматизированных систем играет верификация и оценка соответствия программных средств и программно-технических комплексов информационно-управляющих систем. Вопросам общей теории верификации и оценки соответствия программного обеспечения посвящены работы ряда отечественных и зарубежных специалистов: В. В. Липаева, А. А. Корниенко, М. А. Еремеева, А. Г. Ломако, И. Б. Щубинского, В. В. Куламина, Д. В.

Богданова, В. В. Фильчакова, Г. Майерса, Б. Боэма, Ч. Хоара, Э. Дейкстры и др. Вопросы построения и применения метрической теории программ к проблеме верификации программных средств рассматриваются в своих работах М. Холстед, У. Хансен, Дж. Майерс, Т. Мак-Кейб, З. Чен и другие авторы. Оценивание трудоемкости проектирования и качества разработки программ с использованием метрик сложности проводится с помощью специализированного программного обеспечения, разрабатываемого компаниями Programming Research (PRQA), Scientific Toolworks, MathWorks, Parasoft, McCabe Software, Rational Software, Abraxas Software, IBM, Microsoft, M Squared Technologies, SonarSource, NDepend, ООО "Program Verification Systems" и др.

Существующие процедуры и методы подтверждения соответствия программных средств по требованиям безопасности информации обладают следующими недостатками:

- значительная вычислительная сложность процедур статического и динамического анализа, что является особенно актуальным для архитектурно и функционально сложных информационных систем, и, как следствие, большая трудоемкость работы эксперта;
- недостаточность предписанных нормативными документами проверок, непосредственно связанных с безопасностью программного средства;
- неопределенность действий экспертов и достоверности получаемых результатов при отсутствии точно декларированных способов проведения некоторых предписанных проверок;
- сложность проведения испытаний в отсутствие исходных текстов программ.

Таким образом, задачи совершенствования процедур и методов контроля отсутствия недекларированных возможностей (НДВ) для повышения полноты и сокращения времени выявления НДВ программ

являются актуальными в области подтверждения и оценки соответствия программных средств (ПС) по требованиям безопасности информации.

Объект диссертационного исследования – программные средства при их верификации, подтверждении соответствия по требованиям безопасности информации и анализе защищенности.

Предмет диссертационного исследования – контроль наличия/отсутствия НДВ ПС с использованием метрического анализа.

Цель диссертационного исследования – повышение полноты и сокращение временных затрат на проведение испытаний при контроле наличия/отсутствия НДВ программных средств.

Научная задача исследования состоит в разработке научно-методического обеспечения процедур выявления НДВ ПС на основе анализа семантик, структуризации моделей представления программ и оценивающих их метрик топологической и информационной сложности.

Достижение поставленной цели и решение научной задачи требует решения следующих частных задач:

- системный анализ проблемных вопросов верификации и подтверждения соответствия ПС по требованиям качества и безопасности информации на железнодорожном транспорте;
- обоснование структурированных формальных моделей представления ПС и оценивающих их метрик сложности;
- разработка метода выявления НДВ программных средств на основе структурированных метрик сложности;
- разработка и обоснование методики выявления НДВ ПС на основе структурированных метрик топологической и информационной сложности и разработка практических рекомендаций по использованию метрического анализа при испытаниях на отсутствие НДВ.

Для решения поставленных задач применялись такие методы, как системный анализ, теория верификации программного обеспечения,

формальная логика, аппарат алгебры Янова, методы программометрии. Теоретическую основу и методологическую базу исследования составляют труды отечественных и зарубежных ученых в области формальной верификации, оценки соответствия ПС и программометрии, стандарты в области информационных технологий и информационной безопасности.

На защиту выносятся следующие научные результаты:

- метод выявления дефектов, подобных НДВ, в программных средствах с использованием модели программы в виде схемы Янова и совокупности структурированных метрик сложности;
- методика выявления НДВ программных средств в дизассемблированном коде с использованием структурированных метрик топологической и информационной сложности;
- предложения по построению подсистемы метрического анализа и ее реализации в системе формальной верификации и выявления НДВ программных средств.

Научная новизна работы заключается в следующем:

- предложен подход к выявлению НДВ, базирующийся на анализе формальных семантик, вычислительных структур и свойств программ, структуризации моделей представления программы и оценивающих их метрик сложности;
- разработан метод выявления дефектов, подобных НДВ, основанный на формировании метрического базиса в виде структурированных метрик топологической и информационной сложности, приведении управляющего графа программы к каноническому представлению схемы Янова и продуктивной структуре, построении системы критериев метрического оценивания безопасности функциональных объектов (участков программного кода) на базе результатов экспериментальных исследований;

- разработана методика выявления НДВ программ в дизассемблированном коде с использованием структурированных метрик топологической и информационной сложности и включением подсистем метрического анализа на этапах лексико-синтаксического и структурно-семантического статического анализа.

Достоверность полученных научных результатов определяется корректным применением методов исследования, доказанностью ряда утверждений, лежащих в основе структурированных моделей представления программ и метода выявления НДВ программных средств с использованием структурированных метрик топологической и информационной сложности, аprobацией результатов диссертационных исследований на научно-практических конференциях и их внедрением в организациях.

Практическая значимость результатов исследования состоит в возможности полноты выявления НДВ и сокращении временных затрат при проведении испытаний при контроле отсутствия НДВ программных средств за счет использования разработанных метода и методики выявления недекларированных возможностей программ в дизассемблированном коде с использованием структурированных метрик топологической и информационной сложности.

Практическая ценность и новизна работы подтверждаются актами внедрения, выданными ФГБОУ ВПО ПГУПС, ООО «ЦБИ», ФГУП «ЗашитаИнфоТранс».

Основные результаты исследований излагались и обсуждались на научных семинарах кафедры «Информатика и информационная безопасность» ПГУПС, а также на международной научно-практической конференции «Инфотранс-2007, 2008» (Санкт-Петербург, ПГУПС), на VI международной научно-практической конференции «ТелекомТранс–2008» (Ростов-на-Дону, РГУПС), на III российской конференции с

международным участием «Технические и программные средства систем управления, контроля и измерения» (Москва, Институт проблем управления имени В.А. Трапезникова РАН, 2012), на международной научно-практической конференции «Интеллектуальные системы на транспорте» (Санкт-Петербург, ПГУПС, 2011, 2012).

По результатам исследования опубликовано 10 статей, 3 из которых – в журналах, рекомендуемых ВАК, 5 – в материалах общероссийских, межрегиональных и международных конференций, выпущено 1 учебное пособие, материалы исследования использованы в 2 научно-исследовательских работах, получено свидетельство о государственной регистрации программы для ЭВМ.

Диссертация включает введение, четыре главы, заключение и список использованных источников. Общий объем диссертации – 161 с., из которых основного текста – 139 с. Библиографический список содержит 84 наименования. Основной текст диссертации включает 15 рисунков и 9 таблиц.

В первой главе проводится системный анализ проблемных вопросов верификации и подтверждения соответствия ПС по требованиям качества и безопасности информации на железнодорожном транспорте. Выполняется анализ критериального аппарата, задаваемого стандартами. Приводится общая характеристика методов исследования программ, их верификации и выявления НДВ. Проводится анализ возможностей использования метрик сложности для задач формальной верификации и выявления НДВ программ.

Вторая глава посвящена анализу формальных семантик и структуризации моделей представления программ и оценивающих их метрик сложности. Проводится анализ трактовок и подходов к формальному описанию семантики языков программирования. Осуществляется выбор моделей представления программ на основе

структуризации семантик. Выполняется построение логической модели программы с использованием граф-ориентированных схем Янова.

Третья глава посвящена разработке метода выявления дефектов, подобных НДВ, с использованием структурированных метрик сложности. Сущность метода состоит в формировании метрического базиса как совокупности структурированных метрик топологической и информационной сложности, соотнесенных с конструкциями языка ассемблер; последующем их ранжировании путем экспертного оценивания; приведении логической модели программы к продуктивной структуре, построенной на основе канонического представления схем Янова; экспериментальном определении «чувствительности» и корреляции выбранных компонент метрического базиса к изменениям программного кода, и, на этой базе, разработке системы критериев оценивания безопасности функциональных объектов (участков программного кода).

В четвертой главе разработана методика выявления НДВ программ, определены место и роль метрических оценок в системе статического анализа программных средств. Представлены практические рекомендации по использованию метрик сложности в процедурах подтверждения соответствия программных средств по требованиям качества и информационной безопасности. Также в этой главе представлены результаты экспериментов по оценке разработанного метода верификации с точки зрения оперативности и полноты выявления НДВ.

В заключении сформулированы основные результаты работы, определены рекомендации по их внедрению. Сделан вывод о выполнении поставленных задач и достижении цели исследования.

# **1 СИСТЕМНЫЙ АНАЛИЗ ПРОБЛЕМЫ ПОДТВЕРЖДЕНИЯ СООТВЕТСТВИЯ ПРОГРАММНЫХ СРЕДСТВ ПО ТРЕБОВАНИЯМ БЕЗОПАСНОСТИ ИНФОРМАЦИИ НА ЖЕЛЕЗНОДОРОЖНОМ ТРАНСПОРТЕ**

## **1.1 Тенденции развития системы подтверждения соответствия программных средств по требованиям качества и безопасности на железнодорожном транспорте**

В условиях динамично нарастающей информатизации железнодорожного транспорта для поддержания высокого уровня безопасности движения, грузовых и пассажирских перевозок, корпоративного управления, других критичных технологических процессов и систем важное значение приобретают вопросы обеспечения функциональной и информационной безопасности информационно-управляющих и автоматизированных систем. Их основу составляют программные средства – программируемые микропроцессорные устройства, программное обеспечение, программно-технические комплексы.

Одной из важнейших организационно-правовых мер обеспечения функциональной и информационной безопасности информационно-управляющих и автоматизированных систем является подтверждение соответствия, сертификация и декларирование соответствия программных средств в обязательной и добровольной форме.

Основные понятия и принципы подтверждения соответствия заложены Федеральным законом «О техническом регулировании» № 184-ФЗ от 27.12.2002 г. [1].

Федеральный закон (ФЗ) «О техническом регулировании» регулирует отношения, возникающие при:

- разработке, принятии, применении и исполнении обязательных требований к продукции или к связанным с ними процессам проектирования (включая изыскания), производства, строительства, монтажа, наладки, эксплуатации, хранения, перевозки, реализации и утилизации;
- разработке, принятии, применении и исполнении на добровольной основе требований к продукции, процессам проектирования (включая изыскания), производства, строительства, монтажа, наладки, эксплуатации, хранения, перевозки, реализации и утилизации, выполнению работ или оказанию услуг;
- оценке соответствия.

В основе ФЗ «О техническом регулировании» лежит понятие технических регламентов – документов, которые устанавливают обязательные для применения и исполнения требования к объектам технического регулирования (продукции, в том числе зданиям, строениям и сооружениям или к связанным с требованиями к продукции процессам проектирования (включая изыскания), производства, строительства, монтажа, наладки, эксплуатации, хранения, перевозки, реализации и утилизации). Технические регламенты принимаются в целях защиты жизни или здоровья граждан, имущества физических или юридических лиц, государственного или муниципального имущества; охраны окружающей среды, жизни или здоровья животных и растений; предупреждения действий, вводящих в заблуждение приобретателей. Все содержащиеся в технических регламентах требования носят обязательный характер.

Практической реализацией вопросов технического регулирования является подтверждение соответствия – документальное удостоверение соответствия продукции или иных объектов, процессов проектирования (включая изыскания), производства, строительства, монтажа, наладки,

эксплуатации, хранения, перевозки, реализации и утилизации, выполнения работ или оказания услуг требованиям технических регламентов, положениям стандартов, сводов правил или условиям договоров. Подтверждение соответствия осуществляется в целях удостоверения соответствия продукции и процессов техническим регламентам, стандартам, сводам правил, условиям договоров, содействия приобретателям в компетентном выборе продукции, работ, услуг, повышения конкурентоспособности продукции, работ, услуг на российском и международном рынках, создания условий для обеспечения свободного перемещения товаров по территории Российской Федерации, а также для осуществления международного экономического, научно-технического сотрудничества и международной торговли.

Подтверждение соответствия на территории Российской Федерации может носить добровольный или обязательный характер. Добровольное подтверждение соответствия осуществляется в форме добровольной сертификации. Обязательное подтверждение соответствия осуществляется в форме принятия декларации о соответствии или обязательной сертификации. Порядок применения форм обязательного подтверждения соответствия устанавливается ФЗ «О техническом регулировании».

Добровольное подтверждение соответствия осуществляется по инициативе заявителя, на условиях договора с органом по сертификации на соответствие требованиям стандартов любых видов, системам сертификации, условиям договоров.

Обязательное подтверждение соответствия осуществляется только в случаях, установленных соответствующим техническим регламентом, и на соответствие только техническому регламенту. Объектом обязательного подтверждения соответствия может быть только продукция, выпускаемая в обращение на территории Российской Федерации. Форма и схема обязательного подтверждения соответствия устанавливаются

соответствующим Техническим регламентом. Декларация и сертификат имеют равную юридическую силу и действуют на всей территории РФ. Декларирование соответствия осуществляется по одной из двух форм: принятие декларации на основе собственных доказательств и то же на основе собственных и полученных с участием третьей стороны доказательств. Третья сторона может являться орган по сертификации и (или) аккредитованная испытательная лаборатория (центр).

Нормативная правовая система технического регулирования на железнодорожном транспорте, разработанная на основе Федерального закона «О техническом регулировании» № 184-ФЗ от 27.12.2002 является многоуровневой и включает в себя наряду с соответствующими федеральными законами [43]:

- Технические регламенты «О безопасности инфраструктуры железнодорожного транспорта»; «О безопасности железнодорожного подвижного состава»; «О безопасности высокоскоростного железнодорожного транспорта»;
- национальные стандарты и своды правил;
- корпоративные стандарты ОАО «РЖД» (СТО ОАО «РЖД»), нормы безопасности и подзаконные нормативные правовые акты, регулирующие отношения, связанные с эксплуатацией и обеспечением безопасности движения железнодорожного транспорта.

Основными принципами формирования технических регламентов являлись:

- поддержание существующего высокого уровня безопасности на железнодорожном транспорте;
- преемственность по отношению к действующей системе технического регулирования на железнодорожном транспорте;
- гармонизация с требованиями, установленными в международных и европейских стандартах.

Методологической основой системы устанавливаемых в технических регламентах требований является реализация «нового подхода» - обеспечение приемлемого (максимально допустимого) риска. Техническими регламентами задаются обязательные для выполнения существенные требования безопасности с учетом оценки степени риска в виде функциональных требований, качественно определяющих необходимый уровень безопасности.

Выполнение требований технических регламентов обеспечивается путем реализации положений национальных стандартов и сводов правил, применение которых является добровольным. Для подтверждения выполнения требований каждого технического регламента подготовлены два перечня: перечень национальных стандартов, в результате применения которых на добровольной основе обеспечивается соблюдение требований технического регламента; перечень национальных стандартов, содержащих правила и методы исследований (испытаний) и измерений, в том числе правила отбора образцов, необходимых для применения и исполнения принятого технического регламента.

Для инновационной продукции подтверждение соответствия требованиям безопасности технического регламента допускается проводить в форме обязательной сертификации путем выполнения отдельных положений национальных стандартов и/или сводов правил, а также на основании заключений, полученных от экспертного совета.

Отношения, связанные с эксплуатацией железнодорожного транспорта как единой системы, а также отношения, связанные с обеспечением безопасности движения, должны регламентироваться также на уровне подзаконных нормативных правовых актов и нормативных документов федеральных органов исполнительной власти, таких, например, как Правила технической эксплуатации и другие.

Таким образом, образована единая система обеспечения безопасности на железнодорожном транспорте, объединяющая привычные, и главное – обязательные для всех основные инструкции, корпоративные и национальные стандарты и технические регламенты. Такая структура легко гармонизируется с аналогичными европейскими документами и позволяет создать единое нормативное пространство, снимающее многие искусственно созданные барьеры на пути повышения безопасности и развития новой железнодорожной техники [41].

Техническими регламентами заданы обязательные требования безопасности к программным средствам как элементам функциональных подсистем и составных частей объектов технического регулирования [2 – 4].

Для автоматических бортовых систем железнодорожного подвижного состава, в том числе и высокоскоростного, обеспечивают соблюдение требований информационную и функциональную безопасность. Программные средства, как встраиваемые в технические средства, так и поставляемые отдельно должны сохранять работоспособность после перезагрузок, вызванных сбоями/отказами технических средств, и целостность при собственных сбоях; не должны иметь свойств и характеристик, не описанных в своей документации (недекларированных возможностей); должны быть защищены от компьютерных вирусов, от несанкционированного доступа, от последствий отказов, ошибок и сбоев при хранении, вводе, обработке и выводе информации, возможности случайных изменений. Система управления, контроля и безопасности железнодорожного подвижного состава, в том числе и высокоскоростного, работа тягового привода и другого оборудования при любых неисправностях, ошибках и сбоях программного обеспечения не должны допускать изменений характеристик и режимов работы, которые могут привести к нарушению безопасного состояния высокоскоростного

железнодорожного подвижного состава. Сбой системы управления при исправной работе бортовых устройств безопасности не должен приводить к остановке железнодорожного подвижного состава и к нарушению его проектных характеристик.

Для программируемых устройств железнодорожных подсистем электроснабжения, автоматики и телемеханики, электросвязи и их составных частей должна обеспечиваться безопасность функционирования объектов инфраструктуры железнодорожного транспорта, в том числе и высокоскоростного, и продукции.

Нормативную и правовую поддержку требований технических регламентов обеспечивают корпоративные стандарты ОАО «РЖД», нормы безопасности и другие нормативные правовые акты, включающие, в том числе, и требования к автоматизированным системам управления актово-прецензионной работой, корпоративным автоматизированным системам управления финансами, материальными и трудовыми ресурсами, системам сетевого документооборота и делопроизводства, автоматизированным системам управления организационно-экономическими процессами железнодорожного транспорта. В нормах безопасности для таких систем методическое обеспечение оценки соответствия по требованиям функциональной безопасности в целом основано на использовании системы показателей, устанавливаемых ГОСТ 28195-89 и ГОСТ Р ИСО/МЭК 9126-93 с выделением качественных и количественных показателей безопасности, и использовании системы показателей для оценки в соответствии с серией стандартов ГОСТ Р МЭК 61508, в частности, ГОСТ Р МЭК 61508-3-2007 «Функциональная безопасность систем электрических, электронных, программируемых, связанных с безопасностью. Часть 3. Требования к программному обеспечению». Основными целями и задачами ГОСТ Р МЭК 61508-3-2007 являются определение функций безопасности и уровней полноты безопасности

программного обеспечения; установление требований к модели жизненного цикла безопасности программного обеспечения, которые включают в себя применение мероприятий и методов, ранжированных по уровням полноты безопасности и предназначенных для того, чтобы избегать ошибок и отказов программного обеспечения и принимать необходимые меры при их возникновении; предоставление требований к информации, относящейся к подтверждению безопасности программного обеспечения [62].

В настоящее время в связи с комплексным характером требований к качеству и безопасности информационно-управляющих и автоматизированных систем, сертификация программных средств (ПС) на железнодорожном транспорте осуществляется на добровольной основе в двух основных системах сертификации – по требованиям качества (в Системе сертификации на железнодорожном транспорте) и по требованиям безопасности информации (в Системе сертификации средств защиты информации по требованиям безопасности информации) [24].

При проведении испытаний в Системе сертификации на железнодорожном транспорте на практике в настоящее время для оценки соответствия требованиям функциональной безопасности и качества программных средств железнодорожного транспорта применяется одна из двух существующих в настоящее время систем показателей качества программного обеспечения, базирующаяся на следующих стандартах:

ГОСТ 28195-89 «Оценка качества программных средств. Общие положения», в котором вводится четырехуровневая модель оценки качества программного средства (ПС). В целом ГОСТ 28195-89 содержит рекомендуемый (и, возможно, не вполне полный и корректный) перечень качественных и количественных метрик, и в основном ориентирован на представление разработчика программного средства.

ГОСТ Р ИСО/МЭК 9126-93 «Информационная технология. Оценка программной продукции. Характеристика качества и руководства по их применению», рассматривающий обобщенную модель качества ПО из шести характеристик качества ПО, под каждой из которых понимается набор свойств (атрибутов) программной продукции, по которым ее качество оценивается или описывается. Трудности при его применении вызывает разработка на основе типовой модели, приведенной в стандарте, конкретизированной модели качества рассматриваемого программного средства, включающая метрики и методы оценивания и ранжирования с указанием применимости на стадиях жизненного цикла. Модель ГОСТ Р ИСО/МЭК 9126-93 соответствует, в первую очередь, взглядам пользователя на эксплуатируемое программное средство.

На основе выбранной и адаптированной модели качества экспертами проводятся соответствующие испытания (тестирование, экспертиза программной документации) и с помощью выставляемых экспертных оценок и протоколирования выявляемых в процессе оценивания событий или проявлений ошибок (недостатков, нарушений требований) проводится оценка качества программного средства. В итоге формируется сводная таблица результатов, которая входит в состав протокола испытаний и экспертного заключения.

Отметим, что сертификация по требованиям качества, которая проводится для программных средств железнодорожного транспорта, базируется на модели качества программного обеспечения, ориентированной, в первую очередь, на пользовательские характеристики ПС, без оценивания полной функциональной безопасности и качества разработки.

При проведении сертификации по требованиям безопасности информации выделяются два основных вида сертификационных испытаний программных средств, требования к которым определяются

соответствующими руководящими документами (РД) ФСТЭК (Гостехкомиссии) России:

- на соответствие классу защищенности информации от несанкционированного доступа (НСД), что применяется только для комплексных средств защиты от несанкционированного доступа и межсетевые экранов;
- на отсутствие недекларируемых возможностей (НДВ) – данный вид испытаний применяется, в том числе, и для программных средств железнодорожного транспорта.

Существующий порядок сертификации изделий (продукции), в том числе программных средств железнодорожного транспорта, в целом обеспечивает необходимый уровень безопасности транспорта для жизни людей, здоровья потребителей и охраны окружающей среды, предотвращения вреда имуществу потребителей.

Планирование качества, контроль и испытания программного обеспечения проводятся в процессе его разработки и, в дальнейшем, всего жизненного цикла, определенного в ГОСТ Р ИСО/МЭК 12207-99 «Информационная технология. Процессы жизненного цикла программных средств», с целью оценивания соответствия промежуточных продуктов разработки и конечного программного продукта установленным для них требованиям. В процессе разработки программного обеспечения применяются различные виды контроля и испытаний, которые проводятся с использованием процессов верификации, аттестации, совместного анализа и аудита, также определенных в этом стандарте.

Необходимо выделить следующие особенности проведения оценивания соответствия и сертификации программных средств [39]:

- Программный продукт является сложным, динамично развивающимся объектом, с достаточно частным внесением в него изменений, что существенно затрудняет и усложняет его оценивание;

- Для программных продуктов в настоящее время существует большое количество качественных характеристик с высокой сложностью практического оценивания качества;
- Проведение сертификационных испытаний на этапе динамического анализа или тестирования в реальных условиях эксплуатации является сложным и трудоемким процессом, слабо поддающимся моделированию;
- При сертификации доминирует этап экспертизы документации на программное средство;
- Существующие в настоящее время инструментальные средства оценки качества и распознавания недекларированных возможностей имеют ограниченные возможности и достаточно узкую сферу применения.

## **1.2 Анализ стандартов критериального аппарата и проблемных вопросов подтверждения соответствия программных средств по требованиям безопасности информации**

Основными нормативными документами, определяющими требования для проведения сертификации средств защиты информации по требованиям безопасности информации, являются:

- ГОСТ Р ИСО/МЭК 15408-2000 (части 1 – 3). Критерии оценки безопасности информационных технологий.
- ГОСТ Р 51624-2000. Защита информации. Автоматизированные системы в защищенном исполнении. Общие требования.
- ГОСТ Р ИСО/МЭК 9126-93. Информационная технология. Оценка программной продукции. Характеристики качества и руководство по их применению.
- ГОСТ 28195-89 Оценка качества программных средств. Общие положения.

- ГОСТ Р ИСО-МЭК 12207-99 Информационная технология. Процессы жизненного цикла программных средств.
- ГОСТ Р ИСО/МЭК 12119–2000 Информационная технология. Пакеты программ. Требования к качеству и тестирование.
- ГОСТ Р 50739-95. Средства вычислительной техники. Защита от несанкционированного доступа к информации.
- ГОСТ Р 51275-99. Защита информации. Объект информации. Факторы, действующие на информацию. Общие положения.
  - Руководящий документ. Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации.
  - Руководящий документ. Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации.
  - Руководящий документ. Средства вычислительной техники. Межсетевые экраны. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации.
  - Руководящий документ. Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей.

На основе проведенного анализа критериального аппарата данных нормативных правовых документов можно говорить об отсутствии единой общей универсальной модели оценивания программного средства по требованиям функциональной и информационной безопасности, что, безусловно, создает определенные трудности при проведении испытаний для всех участников процесса.

Гармонизация данных нормативных документов и возможность полноценного применения их при проведении сертификационных испытаний достигается при анализе пересечений множеств показателей качества ПО, его функциональной и информационной безопасности и построении соответствующих моделей оценивания.

При проведении ранжирования указанных в стандартах показателей, характеристик и подхарактеристик качества можно выделить две основных модели качества ПО как основу для проведения сертификационных испытаний на разных этапах жизненного цикла [21].

На этапе разработки и внедрения программного средства основное внимание уделяется внутренним (качество кода и внутренней архитектура) и внешним (безопасность, защищенность и работоспособность ПО) атрибутам качества разработки ПО. Подобная модель задается при проведении сертификационных испытаний в Системе сертификации по требованиям безопасности информации (в соответствии с РД Гостехкомиссии России «Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей»). На основе полученных данных статического анализа возможно применение метрик сложности программного обеспечения для оценивания необходимых показателей качества ПС.

В данной модели качества применяются и оцениваются следующие показатели качества ПС в соответствии с ГОСТ 28195:

1. Показатель надежности ПС:

1.2. Работоспособность – способность программы функционировать в заданных режимах и объемах обрабатываемой информации в соответствии с программными документами при отсутствии сбоев технических средств.

2. Показатели сопровождения:

2.1. Структурность – организация всех взаимосвязанных частей программы в единое целое с использованием логических структур “последовательность”, “выбор”, “повторение”;

2.2. Простота конструкции – построение модульной структуры программы наиболее рациональным, с точки зрения восприятия и понимания образом;

2.3. Наглядность – наличие и представление в наиболее легко воспринимаемом виде исходных модулей ПС, полное их описание в соответствующих программных документах;

2.4. Повторяемость – степень использования типовых проектных решений или компонентов, входящих в ПС.

## 6. Показатели корректности:

6.1. Полнота реализации – полнота реализации заданных функций ПС и достаточность их описания в программной документации;

6.2. Согласованность – однозначное, непротиворечивое описание и использование тождественных объектов, функций, терминов, определений, идентификаторов и т. д. в различных частях программных документов и текста программы;

6.3. Логическая корректность – функциональное и программное соответствие процесса обработки данных при выполнении задания общесистемным требованиям;

6.4. Проверенность – полнота проверки возможных маршрутов выполнения программы в процессе тестирования.

Также в данной модели качества возможно оценивание интегральных подхарактеристик качества ПС в соответствии с ГОСТ Р ИСО/МЭК 9126:

- Защищенность (Security) – атрибуты программного обеспечения, относящиеся к его способности предотвращать несанкционированный доступ, случайный или преднамеренный, к программам и данным;

- Способность к взаимодействию (Interoperability) – атрибуты программного обеспечения, относящиеся к способности его взаимодействовать с конкретными системами (примечание – не нарушая принципов функциональной и информационной безопасности).

При проведении динамического анализа, заложенного в требованиях по уровням контроля 1 – 3, модель качества разработки охватывает также и все показатели функциональной безопасности, т.е. оценивание показателей функциональных возможностей ПО из ГОСТ Р ИСО/МЭК 9126:

- Пригодность (Suitability) – атрибут программного обеспечения, относящийся к наличию и соответствуанию набора функций конкретным задачам;
- Правильность (Accuracy) – атрибуты программного обеспечения, относящиеся к обеспечению правильности или соответствия результатов или эффектов.

На этапе эксплуатации программного средства основной является модель качества ПО в использовании («представление пользователя»), определяющаяся следующими вопросами:

- Имеются ли требуемые функции в программном обеспечении?
- Насколько надежно программное обеспечение?
- Насколько эффективно программное обеспечение?
- Является ли программное обеспечение удобным для использования?
- Насколько быстро переносится программное обеспечение в другую среду?

Таким образом, можно определить модель оценивания соответствия ПО в эксплуатации, основными характеристиками которой выступают следующие показатели качества ПО:

- Функциональность (пригодность и правильность функционирования) – в случае проведения сертификационных испытаний по 4 уровню контроля;

- Практичность;
- Эффективность;
- Надежность в эксплуатации;
- Мобильность.

За исключением функциональности, показатели качества данной модели являются общими для двух стандартов оценки качества ПО (таблица 1.1).

Таблица 1.1. Показатели качества ПО в эксплуатации.

<b>ГОСТ 28195</b>	<b>ГОСТ Р ИСО/МЭК 9126</b>
Надежность (в эксплуатации)	Надежность (в эксплуатации)
• Устойчивость функционирования	<ul style="list-style-type: none"> <li>• Стабильность (Maturity)</li> <li>• Устойчивость к ошибке (Fault tolerance)</li> <li>• Восстанавливаемость (Recoverability)</li> </ul>
Показатели удобства применения	Практичность (Usability)
<ul style="list-style-type: none"> <li>• Легкость освоения</li> <li>• Доступность эксплуатационных программных документов</li> <li>• Удобство эксплуатации и обслуживания</li> </ul>	<ul style="list-style-type: none"> <li>• Понятность (Understandability)</li> <li>• Обучаемость (Learnability)</li> <li>• Простота использования (Operability)</li> </ul>
Показатели эффективности	Эффективность (Efficiency)
<ul style="list-style-type: none"> <li>• Уровень автоматизации</li> <li>• Временная эффективность</li> <li>• Ресурсоемкость</li> </ul>	<ul style="list-style-type: none"> <li>• Характер изменения во времени (Time behavior)</li> <li>• Характер изменения ресурсов (Resource behavior)</li> </ul>
Показатели универсальности	Мобильность (Portability)
• Гибкость	• Адаптируемость (Adaptability)

ГОСТ 28195	ГОСТ Р ИСО/МЭК 9126
<ul style="list-style-type: none"> <li>• Мобильность</li> <li>• Модифицируемость</li> </ul>	<ul style="list-style-type: none"> <li>• Простота внедрения (Installability)</li> <li>• Взаимозаменяемость (Replaceability)</li> </ul>

Таким образом, учитывая повышенные требования к программным средствам, применяемым на железнодорожном транспорте, как к системам управления критически важными объектами или технологическими процессами, защищенность, безопасность и работоспособность ПО необходимо оценивать при сертификации по контролю отсутствия недекларированных возможностей в Системе сертификации по требованиям безопасности информации на основе модели качества ПО при разработке и внедрении [21].

Как было указано выше, при проведении сертификации по требованиям безопасности информации выделяются два основных вида сертификационных испытаний программных средств, требования к которым определяются соответствующими руководящими документами (РД) ФСТЭК (Гостехкомиссии) России:

- на соответствие классу защищенности информации от несанкционированного доступа (НСД) – РД «Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации» и «Средства вычислительной техники. Межсетевые экраны. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации»;
- на отсутствие недекларируемых возможностей (НДВ) – РД «Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей».

Недостатком испытаний первого вида является отставание нормативной базы. Согласно РД, под понятие "средства защиты

"информации" подпадают только комплексные средства защиты от несанкционированного доступа и межсетевые экраны.

Для программных средств общего назначения, включающие подсистему либо функции защиты информации (к данной группе можно отнести практически все программные средства, применяемые на железнодорожном транспорте) рекомендовано проведение оценки по отсутствию недекларируемых возможностей. При этом необходимо отметить, что при определении требований к проведению испытаний на отсутствие НДВ есть как положительные, так и отрицательные моменты. К достоинствам следует отнести требования предоставлять исходный код и документацию, осуществлять контроль над избыточностью (что позволяет исключить некоторые закладные элементы) и наличие определения полномаршрутного тестирования, позволяющего выявить большинство уязвимостей несложных программ. В качестве недостатков можно выделить следующие:

- значительная вычислительная сложность статического и динамического анализа, например, формирование и проверка маршрутов программ являются эффективными только для структурно несложного программного обеспечения, для больших комплексов это проблематично.
- отсутствие или недостаточность проверок, непосредственно связанных с безопасностью программного средства, например, нет механизмов выявления ошибок кодирования, связанных с переполнением буфера, гонками, вызовом функций из чужого адресного пространства;
- неопределенность действий экспертов и достоверности получаемых результатов, например, отсутствуют декларированные способы, нацеленные на построение перечня маршрутов при выполнении функциональных объектов (ветвей);
- невозможность проведения испытаний в отсутствие исходных текстов программ.

## **1.3 Общая характеристика методов исследования программ, верификации и выявления недекларированных возможностей программных средств**

### ***1.3.1 Понятие, цели и задачи верификации***

Согласно определению, приведенному в ГОСТ Р ИСО-МЭК 12207-99, верификация представляет собой подтверждение экспертизой и представлением объективных доказательств того, что конкретные требования полностью реализованы, и проводится после выполнения работ на различных стадиях жизненного цикла. В этой связи вводится понятие процесса верификации – процесса определения того, что программные продукты функционируют в полном соответствии с требованиями или условиями, реализованными в предшествующих работах.

Целями верификации являются:

- проверка соответствия между описанием требований к ПО (техническим заданием), проектными решениями, исходным кодом, пользовательской документацией и реальным функционированием ПО;
- проверка соответствия оформления требований, проектных решений, кода и документации соответствующим стандартам;
- проверка соответствия последовательности и содержания операций, выполненных при создании ПО, соответствующим нормам и стандартам.

В рамках жизненного цикла ПО верификация решает следующие задачи [45]:

- выявление случайных и умышленных дефектов в ПО;
- выявление наиболее критичных и подверженных ошибкам частей создаваемого или сопровождаемого ПО;
- контроль и оценка качества ПО во всех его аспектах;

- предоставление заинтересованным лицам информации о текущем состоянии проекта и характеристиках его результатов;
- предоставление руководству проекта и разработчикам информации для планирования дальнейших работ, а также для принятия решений о продолжении проекта, его прекращении или передаче результатов заказчику.

Выявление дефектов в разработанном ПО, в том числе НДВ, является одной из важнейших задач верификации.

### *1.3.2 Понятие недекларированных возможностей*

Для достижения целей верификации программного средства необходимо четко определить понятие ошибки и/или дефекта программы. Г. Майерс [50] ввел стандартное определение, что «в программном обеспечении имеется ошибка, если оно не выполняет того, что пользователю разумно от него ожидать». Развивая понятие ошибки программы, Я. Соммервиль [82] выделяет частный случай ошибки, когда программа не соответствует своей функциональной спецификации (описанию, разрабатываемому на предшествующему непосредственному программированию этапе) в отдельное понятие – дефект программы. Исходя из определения требований по безопасности информации к современным программным средствам, можно сузить понятие дефекта программы, рассматривая только те из них, при которых нарушаются требования безопасности. Таким образом, наряду с общепринятыми свойствами, характеризующими качество программного обеспечения, на передний план выдвигается отсутствие недекларированных возможностей (НДВ) в программном обеспечении.

Недекларированные возможности ПО, согласно РД НДВ, – это функциональные возможности ПО, не описанные или не соответствующие описанным в документации, при использовании которых возможно

нарушение конфиденциальности, доступности или целостности обрабатываемой информации.

Наличие НДВ порождает уязвимости, которые отрицательно влияют на надежность и устойчивость функционирования систем. В общем случае НДВ подразделяются на:

- случайные – порождаются вследствие ошибок разработчиков;
- преднамеренные – умышленно внедряются в ПО при его разработке. К преднамеренным НДВ относятся проектные, алгоритмические и программные закладки [37]. Именно программные закладки представляют наибольшую опасность, т. к. цель их использования, как правило, связана с нарушением защищенности, при этом закладки являются трудно обнаружимыми.

### *1.3.3 Описание классов и типизация недекларированных возможностей программ*

Система распознавания недекларированных возможностей построена на основании сущности недекларированных возможностей программ. Сущность НДВ раскрывают следующие характеристики: причины появления НДВ, ее цели, возможные последствия, причина срабатывания, демаскирующие признаки.

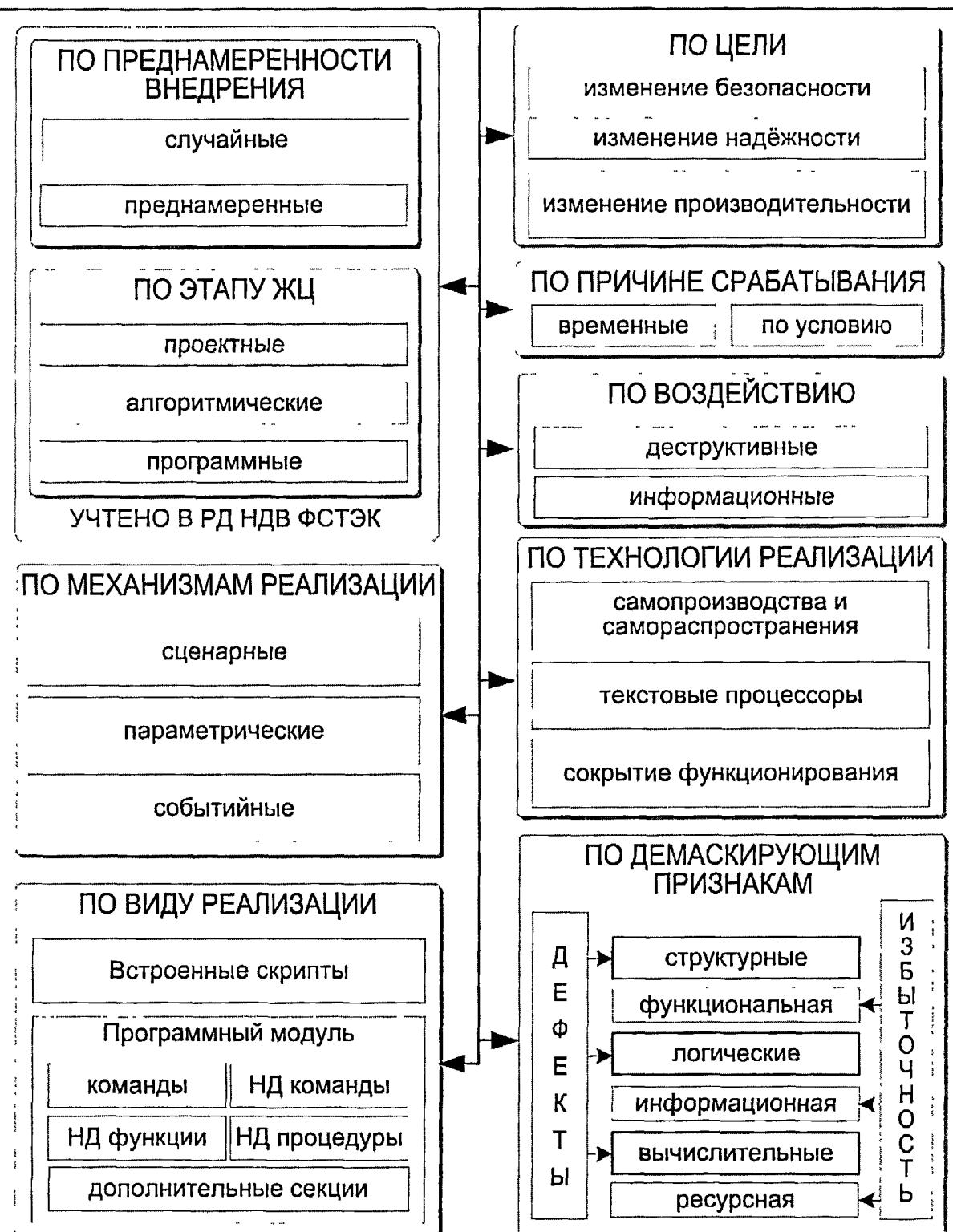


Рисунок 1.1 – Классификация недекларированных возможностей программ

Таким образом, исходя из комплексного представления НДВ программ, можно ввести классификацию НДВ (рисунок 1.1) в соответствии со следующими признаками [63]:

По преднамеренности внедрения:

- случайные или непреднамеренные возможности, выражающиеся в неадекватной реализации разработанного алгоритма и ошибками в реализации;
- преднамеренные НДВ, внедренные с целью преднамеренного несанкционированного нарушения безопасности информации и несанкционированного управления данными, ресурсами и самими системами.

Выделенные типы могут быть внедрены на следующих этапах жизненного цикла программ:

- проектный;
- алгоритмический;
- программный.

По воздействию:

- деструктивные, предназначенные для уничтожения обрабатываемой информации или вывода из работоспособного состояния используемого ПО;
- информационные, предназначенные для подмены или модификации обрабатываемой информации.

По виду реализации:

- основанные на встроенных в программы скриптах;
- основанные на встроенных в программы программных модулях.

Встроенные в программы скрипты реализуют концепцию командного управления операционной системой по определенному сценарию.

Программные модули реализуются посредством:

- команд языка программирования, недекларированных команд процессора или используемого компилятора;
- недекларированных функций и процедур;

- внедрения дополнительных секций в исполняемые файлы и используемые ими библиотек.

Модульная структура программ позволяет реализовывать функции хранения и трансляции новых программ, объединения их с декларированными программными модулями и загрузки в оперативную память.

По механизмам реализации:

- сценарные (или скриптовые) – выполненные в виде вредоносных сценариев (скриптов), написанных на одном из популярных скриптовых языков программирования (VBScript или JavaScript), и входящие в состав исполняемого ПО;
- параметрические – реализованные в виде параметрических процедур как объектов процедурного программирования либо составных программных объектов;
- событийные – реализованные как объекты событийно-ориентированного программирования, т.е. рассматривающие событие как информационный объект, вызывающий реакцию программной системы.

По технологии реализации:

- “вирусная” технология, которая выражается в реализации последовательности действий, приводящих к скрытому копированию в память объекта “заражения”, саморазмножению и выполнению заложенной целевой функции, либо доставке тела вируса использования в виде программной закладки;
- текстовые процессоры распознают ключевые слова в обрабатываемых пользователем текстовых файлах и реализуют новые функциональные связи между декларированными функциями, образуя новые недекларированные сервисы;

- сокрытие функционирования, основанное на технологических приемах сокрытия функционирования программ от операционной системы (предназначено для сокрытия выполнения вычислительных процессов).

По причине срабатывания:

- временные – срабатывание исполнительной части НДВ основано на программно-аппаратных счетчиках;
- условные – срабатывание исполнительной части НДВ происходит с наступлением установленного события.

По цели:

- изменение безопасности;
- изменение надежности;
- изменение производительности.

Изменение безопасности, надежности и производительности можно производить как в сторону улучшения, т.е. повышения безопасности, надежности и производительности, так и в сторону снижения. Снижение безопасности, надежности и производительности посредством НДВ проводиться перед информационным воздействием. Выявление данного класса НДВ является основной задачей сертификационных испытаний средств защиты информации. Тактика, реализующая указанные действия, должна реализовываться в определенном порядке, т.е. последовательное ухудшение безопасности, надежности или производительности должно учитываться как демаскирующий признак срабатывания НДВ программы.

В качестве демаскирующих признаков рассматриваются вычислительные дефекты и избыточность программ.

Избыточность программ будем классифицировать по методам вычисления избыточности программ, вычисляющие:

- функциональную;
- информационную;
- ресурсную избыточность.

Стоит отметить, что заблаговременное вскрытие НДВ применением демаскирующих признаков избыточности программ возможно только при наличии соответствующих эталонов.

Демаскирующие признаки, основанные на вычислительных дефектах программ, подразумевают, что НДВ возможно выявлять как заблаговременно, так и по факту срабатывания НДВ.

#### *1.3.4 Процедуры и методы статического анализа исходных текстов программ*

Простейшие формы статического анализа осуществляются компиляторами. Обычно набор обнаруживаемых ими ошибок ограничен нарушениями синтаксиса и правил совместимости типов. На другом конце спектра систем статического анализа находятся средства автоматического доказательства правильности программ. Эти средства требуют детальной формальной спецификации свойств программы, которые надо доказать, поэтому являются требовательными к ресурсам.

Проведение статического анализа «вручную» подготовленным экспертом заключается в детальном просмотре исходного текста программ. Учитывая большие объемы и сложность построения алгоритмических конструкций современных программных средств применение данного метода выполнения статического анализа приводит к большим затратам времени и человеческих ресурсов и требует привлечения эксперта высокой квалификации и как программиста, и как эксперта качества. При этом точность полученных результатов, как правило, не получается удовлетворительной. В ряде случаев человеческие возможности просто не позволяют провести статический анализ «вручную».

Автоматизированные инструментальные средства способны вычислить или, хотя бы, помочь в определении самых трудоемких характеристик, позволяя таким образом эксперту сосредоточить внимание

на интерпретации полученных характеристик в уровень качества программы [23].

В целом полный процесс статического анализа ПО включает в себя три основных вида исследования исходных текстов программы [42]:

1. Лексический анализ, заключающийся в поиске лексем элементов НДВ (в том числе в шестнадцатеричном представлении), называемых сигнатурой элементов НДВ.

Лексический анализ поверхностно рассматривает исходный код программы и разбивает его на соответствующие токены (tokens). Токен - это значащая часть исходного кода. Примеры токенов включают ключевые слова, пунктуационные знаки, и литералы, такие как числа и строки. Не токены включают пробелы, которые обычно игнорируются, но используются для разделения токенов, и комментарии.

В данном случае осуществляется поиск сигнатур (токенов) следующих классов:

- сигнатуры элементов НДВ;
- сигнатуры «подозрительных функций»;
- сигнатуры штатных процедур использования системных ресурсов и внешних устройств.

Поиск сигнатур реализуется с помощью специальных программ-сканеров.

2. Синтаксический анализ, предполагающий поиск, распознавание и классификацию синтаксических структур НДВ, а также построение структурно-алгоритмической модели самой программы.

Синтаксический анализ заключается в поиске алгоритмических образов основных и дополнительных элементов НДВ в соответствии с принципами концептуального представления НДВ. Во время синтаксического анализа извлекается значение из исходного кода для проверки синтаксической правильности программы (собственно

синтаксический анализ) и построения ее внутреннего представления (синтаксически-структурный или структурный анализ). Решение задач поиска и распознавания синтаксических структур НДВ имеет самостоятельное значение для верификационного анализа программ, поскольку позволяет осуществлять поиск элементов НДВ, не имеющих сигнатуры.

Структурно-алгоритмическая модель программы необходима для реализации следующего вида анализа - семантического.

3. Семантический анализ, заключающийся в поиске НДВ на основе знаний способов и методов организации виртуальных сред, хранящихся в базе данных.

Семантический анализ предполагает исследование программы изучения смысла составляющих ее функций (процедур) в аспекте операционной среды компьютерной системы. В отличие от предыдущих видов анализа, основанных на статическом исследовании, семантический анализ нацелен на изучение динамики программы - ее взаимодействия с окружающей средой. Процесс исследования осуществляется в виртуальной операционной среде с полным контролем действий программы и отслеживанием алгоритма ее работы по структурно-алгоритмической модели. Семантический анализ является наиболее эффективным видом анализа, но и самым трудоемким. По этой причине целесообразно сочетать в себе три перечисленных выше вида анализа. Выработанные критерии позволяют разумно сочетать различные виды анализа, существенно сокращая время исследования, не снижая его качества.

Каждый из видов анализа представляет собой законченное исследование ПО согласно своей специализации. Результаты исследования могут иметь как самостоятельное значение, так и коррелироваться с результатами полного процесса анализа ПО.

Статический анализ исходных текстов программ предполагает выполнение следующих основных процедур:

- предварительный контроль качества разработки программы на основе количественного оценивания выбранных метрик сложности;
- контроль полноты и отсутствие избыточности исходных текстов ПО на уровне функциональных объектов (процедур);
- контроль соответствия исходных текстов ПО его объектному (загрузочному) коду;
- контроль полноты и отсутствие избыточности исходных текстов ПО на уровне файлов;
- контроль связей функциональных объектов (модулей, процедур, функций) по управлению;
- контроль связей функциональных объектов (модулей, процедур, функций) по информации;
- контроль объектов различных типов (глобальных, локальных и внешних переменных, и т.п.);
- формирование перечня маршрутов выполнения функциональных объектов (процедур, функций).

Исходными данными для проведения исследования ПО на НДВ являются:

- программная документация в соответствии с ГОСТ серии 19 (ЕСПД);
- исходные тексты программ;
- библиотечное окружение программ;
- среда разработки ПО;
- среда применения ПО.

К основным этапам статического анализа относятся:

- препроцессорная обработка программы стандартными средствами компилятора;

- лексический и синтаксический анализ;
- формирование базы данных о программных объектах;
- формирование управляющего графа программы;
- формирование визуализаций компонент ПО;
- формирования запросов к базе данных программных объектов и навигации по визуализациям;
- формирование отчета статического анализатора.

Дополнительно выполняются процедуры, позволяющие объединить качественные и количественные показатели статического анализа:

- анализ исполняемых файлов и библиотечного окружения на соответствие списка функций в заголовке и имеющихся в файле;
- формирование перечня дефектов программного кода, дающих суждения о наличии программных закладок;
- формирование полного графа по управлению, распознавание управляющих конструкций *<if - then>*, *<switch>*;
- расчет метрик оценки программного средства, получение метрической оценки управляющего графа и каждой функции (уровень языка, уровень программы, цикломатическое число и др.);
- применение процедуры интерпретации команд языка ассемблер (если необходимо) для получения графа по информации, заключающейся в выделении участков кода, не участвующих в формировании графа по информации, и выделении нетипичных цепочек команд языка ассемблера, подозрительных на НДВ.

Алгоритм работы системы проведения статического анализа исходных текстов ПО на основе рассмотренных методов и процедур представлен на рисунке 1.2. В качестве потенциальных недекларированных возможностей программы рассматриваются выявленные при работе системы статического анализа потенциально «опасные и подозрительные» фрагменты программы, выявляемые

формальными средствами анализа структурных компонентов управляющих графов модулей ПО.

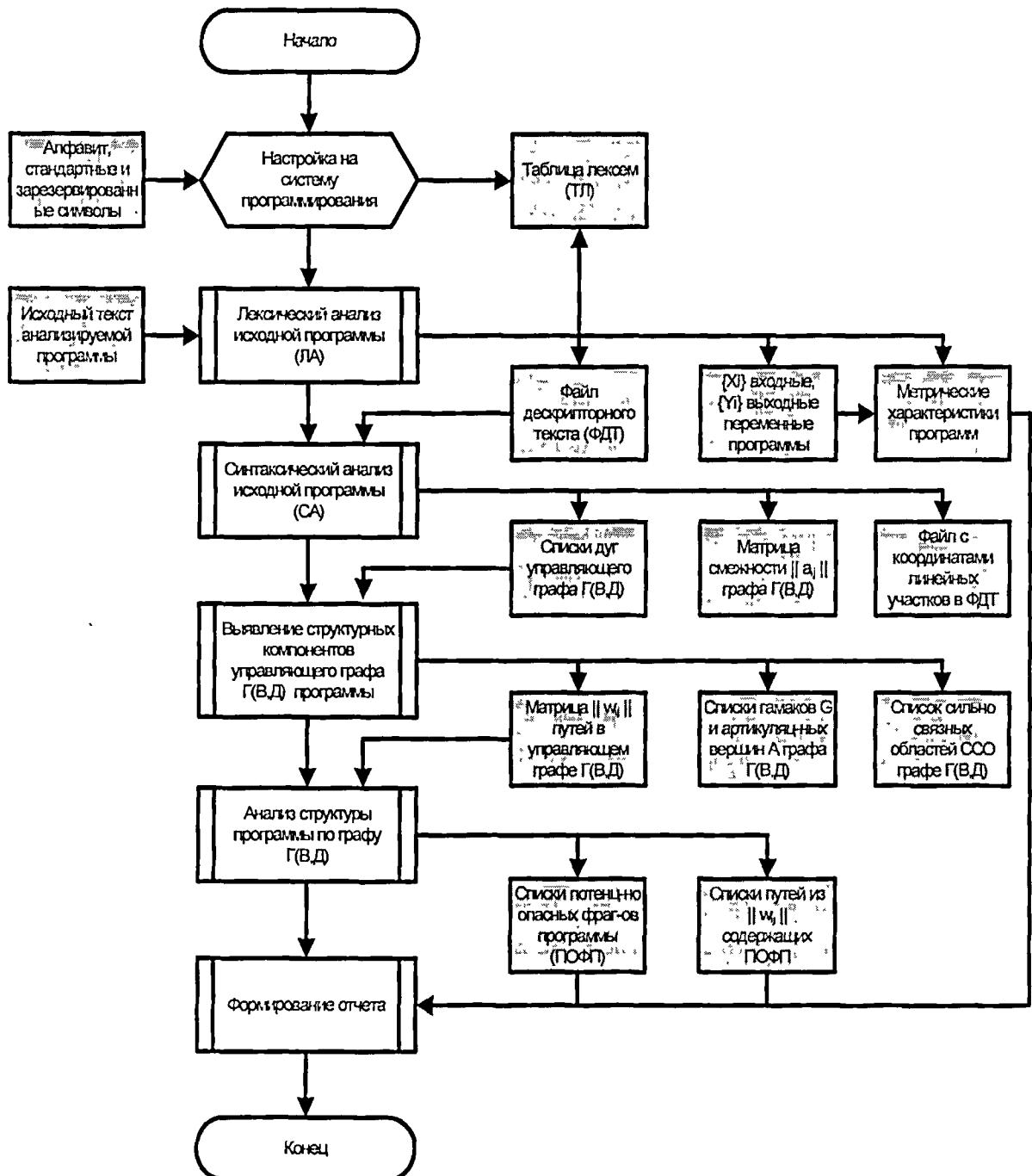


Рисунок 1.2 - Схема алгоритма системы статического анализа

Статическому анализу подвергается синтаксически правильная программа, проверенная с помощью стандартных средств.

Лексический анализатор преобразует исходный текст программы в дескрипторный текст. Цель данной операции – обеспечить унификацию последующих этапов анализа и подготовить для них соответствующие исходные данные.

Синтаксический анализатор строит на основании полученного дескрипторного текста программы управляющий граф программы  $\Gamma(B, D)$ , где  $B = \{b_i\}$  – множество вершин графа (линейных участков программы), а  $D \subseteq \{B \times B\}$  – множество дуг (связей по управлению между линейными участками). Управляющий граф программы  $\Gamma(B, D)$  может быть представлен списком дуг в файле или матрицей инциденций на двух ортогональных с общими элементами списках в основной памяти. Все описанные в программе функции и процедуры представляются отдельными управляющими графиками  $\Gamma(B, D)$ .

Синтаксический анализатор выявляет все структурные компоненты  $\Gamma(B, D)$  и проверяет их на наличие (отсутствие) потенциальных НДВ с помощью процедуры строгого упорядочения  $\Gamma(B, D)$ , а также готовит исходные данные для верификации и динамического анализа.

В процессе выполнения статического анализа должна быть получена следующая информация:

- перечень программных компонентов  $\{K_i\}$  с указанием их связей по управлению и данным;
- управляющие графы  $\Gamma_i$  для компонентов  $K_i \in K$ ;
- перечень возможных путей вызова компонентов  $K_i \in K$ ;
- перечень переменных  $X_i$  и  $Y_i$  компонентов  $K_i$ ;
- перечень неиспользуемых переменных из  $X_i$ ;
- схемы программ  $P_i$  компонентов  $K_i$ ;
- список  $L_i$  неструктурированных фрагментов  $\Gamma_i$ , нарушающих требования структурного программирования и/или неподдающихся строгой упорядоченности нумерацией вершин  $\Gamma_i$ ;

- список структурных фрагментов  $\Gamma_i$ , код которых подлежит углубленному анализу при верификации;
- список гамачных структур  $G_i$  в терминах элементов  $B_i$ ;
- список сильно связных областей  $C_i$  в терминах элементов  $b_{ij} \in B_i$ ;
- список обращений к процедурам  $P_i$  в терминах элементов  $b_{ij} \in B_i$ ;
- список артикуляционных вершин  $A_i$ .

Выявление потенциальных НДВ программы осуществляется в процессе анализа управляющего графа  $\Gamma(B, D)$ . Для проведения анализа графа  $\Gamma(B, D)$ , как правило, используется его представление либо матрицей смежности  $||m_{ij}||$  размером  $|B| \times |B|$ , либо списком инциденций  $L[\cdot]$  размером  $|B| + |D|$ .

При выявлении передач управления проверяется, осуществляются ли они в пределах сегмента кода. Если выявляется выход за длину сегмента, то фиксируется обращение к потенциально возможной программной закладке.

Если в  $\Gamma(B, D)$  обнаружена висячая вершина, то это означает, что имеется участок программы, на который есть передача управления, но нет выхода из него. В этом случае фиксируется потенциальная НДВ программы, которая может являться случайной ошибкой программирования или преднамеренной программной закладкой.

Если в  $\Gamma(B, D)$  выявлены изолированные вершины, то это означает, что в программе имеются участки кода, на которые не происходит передача управления. Такие участки программного кода считаются возможными НДВ программы, как потенциальные программные закладки.

### **1.3.5 Процедуры и методы динамического анализа программного обеспечения**

Целью динамического анализа является контроль соответствия заявленных функциональных возможностей программного средства реально реализованным в программе.

Анализ осуществляется на специально подготовленной версии ПО, в которую встроены специальные программные средства контроля и идентификации (датчики, счетчики и обращения к ним), позволяющие определять маршруты (пути) выполнения заявленных функций ПО с точностью до линейных участков, а также контролировать используемое адресное пространство.

В процессе динамического анализа основное внимание уделяется выявленным при статическом анализе «опасным» или «подозрительным» фрагментам программ. Для этого на этапе их выявления (статического анализа) планируются и формируются специальные тесты. Вместе с тем возможно и принудительное выполнение маршрутов с «опасными» или «подозрительными» фрагментами. Осуществляется это путем выделения текста исходной программы, соответствующего данному маршруту.

На этапе динамического анализа с использованием моделей управляющих структур функциональных объектов решаются следующие задачи:

- разбиение множества линейных участков  $B = \cup B_i, B_i \in \Gamma_i$  на два подмножества  $B^+$  и  $B^-$  соответственно задействованных и незадействованных при выполнении заявленных в программной документации команд (процессов);
- уточнение функционального назначения программных компонентов  $K_i \in K$ , для которых  $B_i+ \neq \emptyset$ ;
- выявление потенциальных НДВ.

Для решения данных задач в исходные тексты исследуемого программного обеспечения встраиваются средства контроля задействования линейных участков  $b_{ij} \in B_i$  – программные закладки исследователя, которые должны сформировать данные, позволяющие выделить из полного образа ПО (множество  $B$ ) потенциальную область поиска НДВ (подмножество  $B_-$ ).

Уточнение функционального назначения программного компонента  $K_i$  можно осуществлять по частоте задействования компонентов  $B_i = \{b_{ij}\}$ .

Выявление потенциальных НДВ проводится на основе адреса структуры, хранящей указатели процессов, отвечающих за выполнение базовых функций ПО, и связи участков в секции данных с компонентами  $K_i$ , соответствующими обновлениям ПО.

В случае, если исходный текст программы представлен объектным модулем, необходима настройка на платформу. В этом случае код программной закладки исследователя должен соответствовать машинному языку. Возможен вариант, когда закладка будет осуществляться после дизассемблирования. Закладка содержит функциональное тело, которое, как правило, одно, и определенную совокупность рассеянных по программе обращений к телу. Обращение (вызов функционального тела) размещается в одной из альтернативных вершин каждой пары, образующих гамачную структуру. Если  $\Gamma(B, D)$  содержит  $n$  путей из входной в выходную вершину, то необходимо внедрить не более  $K = \lceil \text{entire}(\log_2 n) + 1 \rceil$  обращений к телу, где операция  $\text{entire}()$  означает округление.

Динамическое тестирование предполагает прямое тестирование на полноту и корректность выполняемых (заявленных) в программе функций и косвенное тестирование НДВ в выявленных потенциально опасных фрагментах программы. Как при прямом, так и косвенном тестировании необходимо моделировать превентивные атаки. Анализ результатов

динамического тестирования должен осуществляться с привлечением эксперта, так как формальные средства анализа дают только необходимые условия правильности. Достаточные условия правильности должен проверять, если это необходимо, эксперт. После применения инструментальных средств анализа к моменту подключения эксперта к анализу объем анализируемой информации существенно сокращается.

Для проведения динамического анализа исследуемый программный комплекс подвергается специальной доработке, при этом эксперт может не только оценивать, насколько данная доработка обеспечивает полноту отработки алгоритма, реализованного в исследуемой программе, но и активно влиять на этот процесс. Суть доработки заключается в формировании на языке разработки программы исследуемого комплекса обращения к некоторой служебной подпрограмме – датчику и внедрении этой конструкции в соответствующие места исходных текстов программ комплекса. В качестве параметров подпрограммы указывается информация, идентифицирующая место в исходной программе, в котором требуется фиксировать факт получения управления в процессе тестирования комплекса.

Реализация алгоритма служебной подпрограммы-датчика сводится к фиксации факта обращения к ней и записи входных параметров на внешний носитель информации для последующей статистической обработки.

Подготовленная версия исходных текстов комплекса с вставленными подпрограммами – датчиками компилируется обычным порядком, и полученная версия комплекса в дальнейшем используется при проведении динамического тестирования.

В процессе проведения динамического анализа должны быть проведены следующие проверки:

- контроль выполнения функциональных объектов;

- построение маршрутов вызовов функциональных объектов;
- сопоставление фактических маршрутов выполнения функциональных объектов (системных вызовов, функций, ветвей) и маршрутов, построенных в процессе проведения статического анализа;
- анализ используемых и неиспользуемых переменных.

Для проведения динамического анализа необходимы исходные тексты программных модулей, входящих в состав ПО, и набор статических характеристик, полученных в процессе предварительного изучения и статического анализа программных модулей и их взаимодействия:

- модульная структура ПО (граф вызовов; матрицы вызовов и достижимости; местоположения (точки) вызовов);
- логическая структура каждого программного модуля (граф по управлению, пути тестирования);
- функциональные спецификации модулей.

Порядок проведения динамического анализа следующий:

- 1) На основании функциональных спецификаций анализируемой программы разрабатывается набор тестов либо вручную, либо с помощью автоматизированных средств подсистемы генерации тестов.
- 2) Тестируемая программа размечается (маркируется), как правило, с помощью специальных программных модулей – датчиков таким образом, чтобы при ее выполнении фиксировался факт прохождения соответствующей ветви.
- 3) Тестируемая программа выполняется на подготовленном наборе тестов. При этом фиксируются выходные данные, полученные при выполнении программы, и информация о покрытии (выполнении) элементов структуры программы (трасса выполнения).
- 4) Проверяется правильность выходных данных (по спецификации). Если выявляется отклонение значений полученных выходных данных от ожидаемых, то фиксируется наличие ошибки. При этом в качестве

параметров ошибки выступают тест, при выполнении которого обнаружена ошибка, и полученный ошибочный результат. После чего тестирование либо продолжается, либо прекращается, в зависимости от требований.

5) Проверяется полнота набора тестов на основе определения степени покрытия элементов структуры тестируемой программы в соответствии с выбранным критерием.

6) Если заданная полнота тестирования не достигнута, то разрабатывается дополнительный набор тестов, ориентированных на покрытие непокрытых элементов структуры тестируемой программы. После этого сценарий тестирования продолжается с шага 3.

В процессе выполнения программы фиксируется время выполнения отдельных функций и сравнивается с заданным значением в спецификации требований (техническом задании). Для оценки времени восстановления моделируются соответствующие сбойные ситуации.

При подготовке тестов следует учитывать, что показатель полноты тестирования проекта коррелирует с показателем работоспособности программы только при условии достаточного уровня качества тестов, адекватных заданным функциональным спецификациям программы. Действительно, обработка программы без ошибок, даже при полноте тестирования равной 100%, на тестовых данных, не соответствующих особенностям реальных данных, не доказывает, что на реальных данных программа также будет работоспособна.

В зависимости от требований, предъявленных к анализируемой программе и характеризующих ее критичность, применяются следующие критерии контроля покрытия структуры программы:

1) покрытие всех ветвей (логических блоков) графа управления программы;

2) покрытие всех достижимых пар комбинаций ветвей;

3) покрытие всех реализуемых путей в графе программы.

### ***1.3.6 Распознавание НДВ по результатам статического и динамического анализа***

В результате проведения статического и динамического анализа в исследуемом программном обеспечении с помощью автоматизированных средств формируется перечень потенциальных НДВ, т.е. фрагментов программного кода (программных конструкций), для которых получены неудовлетворительные результаты соответствия правилам программирования, алгоритмизации и заявленной функциональности программного средства.

По окончании проведения статического и динамического анализа ПО квалифицированным экспертом выполняется сравнение результатов статического и динамического анализа, анализ и обработка полученных данных с целью принятия решения по каждой выявленной потенциально опасной конструкции. В том числе, при проведении экспертного анализа выноситься суждение о классификации каждой выявленной НДВ для следующих целей:

- принятие решения о преднамеренности внесения НДВ в программное средство;
- вынесение рекомендаций разработчикам программного средства по устранению выявленной НДВ при определении ее как непреднамеренной;
- информирование ФСТЭК России о реализации в исследуемом программном средстве программной закладки или вредоносного кода в случае определения выявленной НДВ как преднамеренной;
- создание базы данных выявленных НДВ для формирования эталонных НДВ.

## **1.4 Анализ возможностей использования метрик сложности для задач формальной верификации и выявления недекларированных возможностей программ**

В настоящее время в мировой практике используется несколько сотен показателей, в той или иной степени описывающих сложность программ. Однако при реальном использовании различных метрик исследователи часто пытались максимизировать универсальность применяемых метрик без учета области применения исследуемого программного обеспечения, что существенно снизило доверие разработчиков и пользователей программного обеспечения к метрикам сложности программ как эффективному инструменту оценивания свойств программы на различных этапах жизненного цикла. Подобные обстоятельства требуют тщательного отбора методик, моделей, методов оценки программного обеспечения, установления порядка их совместного использования, применения избыточного разномодельного исследования одних и тех же показателей для повышения достоверности текущих оценок, накопления и интеграции разнородной метрической информации для принятия своевременных производственных решений.

Применение метрического анализа программ на различных этапах жизненного цикла используется в современной практике, в первую очередь, для качества разработки программ, их отдельных качественных показателей, а также для оценивания трудоемкости проектирования и программирования. Вычисление значений отдельных метрик и расчет некоторых интегральных показателей проводится с помощью специализированных программных средств, основные из которых представлены в таблице 1.2. Практически все существующие на сегодняшний день программные продукты ориентированы на анализ исходных кодов программ, написанных на одном конкретном языке высокого уровня, чаще всего С и С++. Количество рассчитываемых в

данных средствах метрик сложности достаточно мало и включает в себя наиболее простые для вычисления метрики. Основным недостатком при этом является отсутствие четко формализованных методик применения метрик для достижения конкретно поставленных целей оценивания.

Таблица 1.2. Основные инструментальные программные средства для проведения анализа исходного кода и расчета метрик сложности.

Компания-разработчик	Программный комплекс	Поддерживаемые языки программирования
Programming Research (PRQA)	<ul style="list-style-type: none"> <li>• QA·Verify</li> <li>• QA·C</li> <li>• QA·C++</li> </ul>	C, C++
Scientific Toolworks	Understand ( <i>Source Code Analysis &amp; Metrics</i> )	Ada, C, C++, C#, COBOL, CSS, Delphi, Fortran, HTML, Java, JavaScript, Jovial, Pascal, PHP, PL/M, Python, VHDL, XML
MathWorks	Polyspace® products	C, C++, Ada
McCabe Software	McCabe IQ	Ada, ASM86, C, C#, C++.NET, C++, JAVA, JSP, VB, VB.NET, COBOL, FORTRAN, Perl, PL1
Abraxas Software	CODECHECK ( <i>C/C++ Source Code Analysis</i> )	C, C++
IBM	Rational ClearCase	
Microsoft	Visual Studio 2008, 2010	C, C#, C++.NET, C++, VB, VB.NET
M Squared Technologies	Software Source Code Metrics and Source Code Analysis Tool	C, C++, C#, Java
SonarSource	Sonar	C, C#, Flex, Natural, PHP, PL/SQL, Cobol, VB 6

NDepend	NDepend	C, C#, C++, VB
ООО "Program Verification Systems"	PVS-Studio	C, C++
<i>свободно распространяемое ПО</i>	C and C++ Code Counter (CCCC)	C, C++

Основой для построения метрической теории программ (программометрии или программометрики) является зависимость операторов от операндов, доказанная М. Холстедом и развитая в работах У. Хансеном, Пивоварским, Дж. Майерсом, Т. Мак-Кейбом, З. Ченом и другими [11 – 13, 32, 50, 68, 80]. Проведённые исследования показали, что применение метрик сложности позволяет проводить оценивание качества программ в различных аспектах, в том числе и по требованиям безопасности информации, и на различных этапах жизненного цикла программного средства. Для рассмотрения подобных возможностей метрик сложности, учитывая их многообразие, необходимо предварительно провести их упорядочивание и классификацию.

По их функциональному назначению метрики программ можно разделить на шесть условных групп:

- Метрики для оценивания топологической и информационной сложности программ;
- Метрики для оценивания надежности программных систем, позволяющие прогнозировать отказовые ситуации;
- Метрики для оценивания производительности программного средства и повышения его эффективности путем выявления ошибок проектирования;
- Метрики для оценивания уровня языковых средств программирования и их применения;

- Метрики для оценивания трудности восприятия и понимания программных текстов, ориентированные на психологические факторы, существенные для сопровождения и модификации программ;
- Метрики для оценивания производительности труда программистов, предназначенные для прогнозирования сроков разработки программ и планирования работ по созданию программных комплексов.

Другой подход определяет три категории метрик сложности, базируясь на структуре и способах построения программного обеспечения. Первая категория определяется как словарные метрики, основанная на метрических соотношениях Холстеда, цикломатических мерах Мак-Кейба и измерениях Т. Тейера. Вторая категория ориентирована на метрики связей, отражающих сложность отношений между компонентами системы, например, метрики Уина, Винчестера, Д. Кафура. Третья категория включает семантические метрики, связанные с архитектурным построением программ и их оформлением - метрики С. Мак-Клуга и Дж. Колофелло. Третья категория метрик затрагивает операционный, аксиоматический и денотационный подходы к формализации программного обеспечения.

Согласно другой классификации, показатели сложности делятся на две группы: сложность проектирования и трудоемкость разработки в целом и сложность выполнения (функционирования) программного средства [32].

Сложность и трудоемкость разработки программного продукта может оцениваться двумя способами:

- по интегральным характеристикам сложности, которые определяются по внешним параметрам программы, не учитывающим ее внутреннюю структуру;
- по структурным характеристикам сложности, учитывающим внутреннюю структуру программы и зависящим от сложности маршрутов

(потоков) управления, сложности потоков данных или специальных свойств графа управления.

Динамическая (или вычислительная) сложность характеризует процесс выполнения программы и имеет три взаимосвязанных составляющих:

- временную – определяется временем выполнения программы или временем ее реакции на запрос пользователя;
- программную – определяется составом и способом взаимодействия процедур или модулей, образующих программу, а также возможностью их размещения в кеш-памяти, основной памяти или на диске;
- информационную – определяется сложностью организации данных и доступа к ним, а также возможностью их размещения в кеш-памяти, основной памяти или на диске.

Основой для построения и развития метрической теории программ явился проведенный М. Холстедом анализ зависимости операторов от операндов и разработанная на базе него система метрик [68].

На основании исследований количественных характеристик текстов в лингвистике уже относительно давно установлен ряд эмпирических закономерностей. Одна из них, так называемый закон Ципфа, является зависимостью между частотой появления тех или иных слов в тексте (выбранных из словаря текста) и его длиной. В конце 70-х годов аналогичные результаты были получены М. Холстедом при изучении текстов программ, написанных на различных алгоритмических языках. Также чисто эмпирически им было найдено соотношение, связывающее общее количество слов в программах с величиной их словарей. Как оказалось впоследствии, эти закономерности могут быть получены и осмыслены теоретически на основе представлений алгоритмической теории сложности.

Если считать, что словарь любой программы (соответствующий алфавиту последовательности) состоит только из имен операторов и operandов, то тексты программ всегда удовлетворяют следующим условиям:

Маловероятно появление какого-либо имени оператора или операнда много раз подряд. Как правило, языки программирования позволяют построить такую конструкцию, в которой подобные фрагменты программы имели бы минимальную длину;

Циклическая организация программ исключает многократное повторение какой-либо группы операторов и operandов;

Блоки программ, требующие периодического повторения при ее исполнении, обычно оформляются как процедуры или функции, так что в тексте программы присутствуют лишь их имена;

Имя каждого операнда должно появляться в тексте программы хотя бы один раз.

Следовательно, каждая программа может рассматриваться как результат сжатия ее развертки (с раскрытыми циклами, подстановкой в нужных местах функций, процедур, макрокоманд и т.п.) Поэтому, используя терминологию алгоритмической теории сложности можно утверждать, что длина программы есть мера сложности развертки и ее текст не может быть существенно сокращен, так как все закономерности (периодичность) развертки уже использованы для этой цели.

Таким образом, текст программы, независимо от его семантики, представляет собой внешне случайную последовательность слов (operandов и операторов), составляющих словарь программы. Поэтому представляется естественной идея поставить в соответствие процессу написания программы некоторый генератор случайных последовательностей.

Теоретически было получено первое соотношение Холстеда как математическое ожидание длины текста программы

$$M(Q_r) = \ln 2\eta \cdot \eta \log 2\eta \approx \eta \log \eta , \quad (1.1)$$

представляющее собой математическое ожидание количества слов в программе, если словарь программы состоит из  $\eta$  слов. Ввиду практической и теоретической важности этого соотношения оно было подвергнуто тщательному статистическому анализу. Специально разработанные программные средства позволили в автоматическом режиме для многих тысяч программ проверить зависимость между количеством слов  $\eta$  и длиной программы  $N$ . Оказалось, что более точным является выражение

$$N = 0,91\eta \log \eta . \quad (1.2)$$

Таким образом, длина программы, представляющая собой математическое ожидание количества слов в тексте программы при фиксированном словаре, является исходной величиной для расчета остальных ее характеристик. Ясно, что программы, реализующие один и тот же алгоритм, написанные разными программистами, будут несколько различаться длиной. Фундаментальное значение формулы Холстеда заключается в том, что это различие с ростом словаря программы весьма быстро уменьшается.

Другая важная характеристика программы - ее объем  $V$ . В отличие от длины  $N$  он измеряется не количеством слов, а числом двоичных разрядов. Если в словаре имеется  $\eta$  слов, то для задания номера любого из них требуется минимум  $\log \eta$  бит. Следовательно, объем программы определяется как

$$V = N \log \eta = \eta \log^2 \eta . \quad (1.3)$$

Если алфавиту поставить в соответствие словарь, а произвольной последовательности символов - последовательность слов в программе,

трактуемой как результат выборки из генеральной совокупности, то с точностью до обозначений полученные соотношения

$$N \approx s \log s, \min H \max = s \log^2 s, N \approx \eta \log \eta, V = \eta \log^2 \eta$$

совершенно идентичны, хотя смысл их различен. В первом случае при фиксированной (заданной) длине  $N$  была определена величина словаря; во втором - решена обратная задача: по заданному словарю найдена длина программы. Идентичность этих выражений говорит о том, что соотношение между величиной словаря и длиной текста единствено и взаимно однозначно. Во-вторых, вероятностная модель текста программы, основанная на наглядном представлении выборки из генеральной совокупности с возвратом, равносильна формальному подходу алгоритмической теории сложности.

Выше было отмечено, что словарь программы состоит только из операторов и operandов. Если количество первых обозначить за  $\eta_1$ , а вторых -  $\eta_2$ , то  $\eta = \eta_1 + \eta_2$  и соотношение Холстеда примет вид

$$N = \eta_1 \log \eta + \eta_2 \log \eta \approx \eta_1 \log \eta_1 + \eta_2 \log \eta_2 = N_1 + N_2. \quad (1.4)$$

Ясно, что величины  $\eta_1$  и  $\eta_2$  независимы и могут принимать произвольные значения. Однако этого нельзя сказать относительно  $N_1$  и  $N_2$ , то есть числа всех операторов  $N_1 = \eta_1 \log \eta$  и всех operandов  $N_2 = \eta_2 \log \eta_2$  в тексте программы: между ними можно установить приблизительное взаимно-однозначное соответствие. Действительно, каждый operand входит в текст, по крайней мере, хотя бы с одним оператором - например, с разделителем (запятой), отделяющим его от других операторов. В то же время применение нескольких операторов к одному operandу маловероятно. Поэтому, можно утверждать, что  $N_1 \approx N_2$ , хотя величины словарей  $\eta_1$  и  $\eta_2$  могут сильно отличаться друг от друга. Это позволяет прийти к весьма важному практическому выводу:

$$N \approx 2N_2 = 2\eta_2 \log \eta_2. \quad (1.5)$$

Путем дальнейшего развития мер оценивания сложности информационного содержания программы был получен еще ряд метрик системы Холстеда:

1) Уровень реализации программы – метрика характеризует степень компактности программы путем сравнения реального объема исследуемой программы с минимально возможным объемом программы для данного алгоритма:

$$L = \frac{V^*}{V} \quad (1.6)$$

2) Интеллектуальное содержание – метрика характеризует интеллектуальную целостность исследуемых процедур программы и вычисляется по формуле

$$I = \frac{V}{D} = \left( \frac{\eta_1^*}{\eta_1} \times \frac{\eta_2}{N_2} \right) \times (\log_2 \eta \times (\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2)) \quad (1.7)$$

где  $D = (\eta_1 N_2) / (2 \eta_2)$  - трудоемкость кодирования.

3) Уровень языка – характеризует качество дизассемблирования, позволяет обнаружить вставки на другом языке, что означает подозрение на наличие в программе закладок:

$$\lambda^* = \frac{V}{D^2} \quad (1.8)$$

4) Оптимальная модульность, позволяющая судить об информационной избыточности программы:

$$M = \frac{\eta_2^*}{6} \quad (1.9)$$

Таким образом, класс так называемых словарных метрик, которой базируется на системе метрик Холстеда, включающий метрики топологической и информационной сложности, может применяться для целей статического анализа программ в силу своей основанности на формализованных структурных свойствах программного обеспечения.

## **1.5 Постановка научной задачи исследования**

На основании проведенного анализа существующих подходов к процедурам подтверждения соответствия программных средств по требованиям безопасности информации и методов их реализации, можно определить, что задачи совершенствования процедур и методов контроля отсутствия недекларированных возможностей для повышения полноты и сокращения времени выявления НДВ программ являются актуальными в области подтверждения и оценки соответствия программных средств по требованиям безопасности информации.

Научная задача данного диссертационного исследования состоит в разработке научно-методического обеспечения процедур выявления НДВ ПС на основе анализа семантик, структуризации моделей представления программ и оценивающих их метрик топологической и информационной сложности.

Достижение поставленной цели и решение научной задачи требует решения следующих частных задач:

- Проведение системного анализа проблемы оценки и подтверждения соответствия программных средств по требованиям качества и безопасности информации на железнодорожном транспорте. Данная задача подробно рассматривается в главе 1 диссертационного исследования.

- Разработка и обоснование структурированных моделей представления программного средства и требований к контролю отсутствия недекларированных возможностей. Анализ существующих подходов к решению данной проблемы, разработка и обоснование предлагаемых в рамках данного исследования структурированных моделей представления программного средства и требований к контролю

отсутствия недекларированных возможностей подробно излагаются в главе 2.

- Разработка метода выявления НДВ программных средств на основе структурированных метрик топологической и информационной сложности, включающего в себя построение метрического базиса как совокупности структурированных метрик топологической и информационной сложности, соотнесенных с конструкциями языка ассемблер; последующее их ранжирование путем экспертного оценивания; приведение логической модели программы к продуктивной структуре, построенной на основе канонического представления схем Янова; экспериментальном определении «чувствительности» и корреляции выбранных компонент метрического базиса к изменениям программного кода, и, на этой базе, разработке системы критериев оценивания безопасности функциональных объектов (участков программного кода) – подробное описание данного метода приводится в главе 3.

- Разработка и обоснование методики выявления НДВ программных средств на основе структурированных метрик топологической и информационной сложности, разработка практических рекомендаций по использованию структурированных метрик сложности при испытаниях на отсутствие НДВ. Данная задача решается в главе 4.

## **2 ОБОСНОВАНИЕ ПОДХОДА К СТРУКТУРИРОВАНИЮ МОДЕЛЕЙ ПРЕДСТАВЛЕНИЯ ПРОГРАММ И МЕТРИК СЛОЖНОСТИ НА ОСНОВЕ АНАЛИЗА СЕМАНТИК**

### **2.1 Анализ трактовок и подходов к формальному описанию семантики языков программирования**

Структурирование моделей представления программ и упорядочение метрик сложности для последующего выбора метрического базиса оценивания безопасности программного средства будем осуществлять на основе анализа методов теоретического программирования, связанных с моделированием и изучением семантики языков программирования.

Формальное описание семантики языков программирования в настоящее время базируется на трех основных трактовках (подходах):

- операционной трактовке;
- аксиоматической трактовке (подходе), основанной на использовании аксиоматического метода или пропозиционной семантики;
- денотационной трактовке (функциональном подходе или подходе, основанном на математической семантике).

Представленные трактовки и подходы семантического описания эквивалентно отражают смысл конструкций языка на своем, отличном от других уровне абстракции. Различие данных подходов заключается в различном выражении семантики вычислимой функции. Вопросам соотношения методов семантического описания уделено достаточное внимание в работах [58, 61, 74, 76, 77, 83, 84] и других. Их основной вывод состоит в утверждении эквивалентности языкового выражения вычислимости в операционной, аксиоматической и денотационной семантиках. Названные подходы не исключают, а дополняют друг друга, позволяя исследовать семантику языков программирования с разных сторон. Обоснованно появилась концепция взаимосвязанных семантик (Ч.

Хоар, Д. Донаху [26, 77]), призванная ослабить смысловые разнотечения авторов языков, разработчиков трансляторов и программистов.

Для любого из приведенных подходов используются общие теории и методы исследования (логические формальные теории, теория алгоритмов, теории множеств, вероятностей, информации и другие), составляющие математические основы исследования программ.

Рассмотрим детальнее сущность различных трактовок семантик программ (языков программирования).

При операционном подходе семантика программы описывается в терминах некоторой машины (автомата), чаще всего – воображаемой (абстрактной) и предполагает задание конкретной последовательности состояний (состояние характеризуется совокупностью значений переменных), включая начальное и множество заключительных состояний, и функций перехода между ними. Поэтому ориентация этой семантики проявляется на этапах построения и реализации вычислимой функции. Эта форма семантики использовалась (в различных вариантах) для определения практически всех императивных языков программирования (Алгол-68, абстрактные автоматы В.М. Глушкова, "модель-вычислитель" МакКарти и другие [57]). Фактически этот подход реализуется всякий раз, когда создается «базовый» интерпретатор языка, служащий эталоном устойчивости действий команд любой другой реализации. Например, при создании одного из первых интерпретаторов - LISP-интерпретатора, написанного Дж. МакКарти [81, 57]. При этом существуют известные трудности данного подхода в доказательстве свойств вычислимости и абстрактной спецификации вычислимой функции.

Аксиоматический метод или пропозиционная семантика – это направление формализации, базирующееся на определении и формулировке утверждений о свойствах состояний и вычислимости, таких, как эквивалентность, частичность (частичная эквивалентность) и

завершаемость. Значение конструкций языка в таких семантиках выражается в терминах множества формул, описывающих состояния объектов программы. В результате применения данного метода строится формальная теория «исчисления программ» – продукционная система, основанная на формулах, выражающих некие утверждения о состоянии программ и правила для вывода программ. Основными методами, используемыми в аксиоматическом подходе, являются методы индуктивных утверждений К. Флойда и П. Наура, К. Хоара, Н. Вирта [16, 53, 76, 77], основанные на построении индуктивных утверждений данных и конструкций программы и на их основе доказательства теоремы верификации, т.е. исследовании свойства корректности программы; метод преобразования предикатов Э. Дейкстры [19], базирующийся на формальном исследовании текста программы с помощью предикатов первого порядка и декларирующий свойство эквивалентности; метод рекурсивных индукций Дж. Маккарти [46], состоящий в структурной проверке функций, работающих над структурными типами данных, структур данных и диаграмм перехода во время символьного выполнения программ и утверждающий свойство завершаемости; обобщенный аксиоматический метод Хоара [6, 75, 78], основанный на модели вычислений, оперирующей с историями результатов вычислений программы, анализом путей прохождения и правил обработки большого объема информации, и моделирующей отношения и причинно-следственные связи, возникающие между операторами языка программирования и, таким образом, доказывающий наличие свойств частичной эквивалентности и завершаемости.

Таким образом, пропозиционная семантика ориентирована на процессы эквивалентных преобразований и доказательство свойств вычислимости (алгоритмической разрешимости), что можно отождествить с одноименными этапами задач синтеза.

Денотационный (или функциональный – по Дж. Бэкусу) подход, получивший также название подхода, основанного на математической семантике, базируется на модели (интерпретации алгоритма), описываемой более традиционными для математики средствами – некоторой совокупностью множеств, отношений и отображений. По замыслу эти средства должны описывать результаты исполнения программы в целом и ее отдельных частей, точнее – связь между начальным и заключительным состоянием, тем самым подняв строгость описания семантики реальных языков программирования и приближая абстрактные машины теории вычислимости к реальным компьютерам. В данном направлении необходимо отметить теорию лямбда-исчислений А. Черча [10, 73], теорию регулярных выражений С. Клини (Клейни) [35, 36]; функциональный стиль программирования Дж. Бэкуса [14, 15, 67], основанный на использовании комбинированных форм для создания программ, включающий типы данных для функционального программирования, алгебраические преобразования программ и оптимизацию; язык РЕФАЛ, созданный В. Турчиным в качестве метаязыка для описания семантики других языков на основе нормальных алгоритмов Маркова [64 – 66]; теории вычислимых нумераций и топологических пространств Ю. Ершова [30, 31] и теорию функциональных областей (пространств) Д. Скотта [61], получившие дальнейшее развитие в теории семантического программирования (теории денотационной семантики); теорию неподвижной точки программ З. Манны [51].

Теоретическим фундаментом, на основе которого развивались представленные теории, явилось понятие рекурсивной функции и основополагающее утверждение о совпадении класса рекурсивных функций с классом вычислимых.

Результаты направления обогатили, главным образом, теорию и воплотились в ряде специальных языков LISP, APL, ML, РЕФАЛ, Haskell и других, нашедших применение к задачам символьной обработки текстов, представления сценарийных знаний, проектирования функциональных структур, доказательства свойств рекурсивных языков и программ, решения проблем, связанных с искусственным интеллектом. Формализация семантики здесь состоит в том, что смысл вычислений задается совокупностью потенциально допустимых вычислительных функций, отображающих всевозможное множество состояний процесса вычислений в себя, в то время как традиционное определение смысла вычислений (операционная трактовка) задается правилом вычисления последовательности состояний программы и функций перехода между ними.

Денотационная трактовка смысла обладает значительными преимуществами при описании функциональной структуры и может эффективно использоваться для моделирования задачи вычислений и ее последующей декомпозиции. Для жизненного цикла программы – это подпроцессы спецификации, проектирования и верификации.

Названные трактовки и подходы дополняют друг друга, отражая семантику языков программирования с разных сторон. Операционный подход отражает содержательную сторону программирования (так называемый «фон-неймановский» взгляд на программирование). Он ближе других стоит к задачам разработки, как интерпретаторов языков программирования, так и компиляторов для них. В последнее время многими исследователями и разработчиками приоритет отдается функциональному, денотационному подходу (Дж. Бэкус и другие).

Наиболее подходящими для решения поставленной научной задачи представляются денотационный и аксиоматический подходы с

использованием соответствующих методов доказательного и верификационного исследования.

Эти подходы позволяют рассматривать концепцию упорядоченных семантик, что делает возможным обосновать ряд структурированных моделей представления программы и соответствующих им структурированных групп метрик сложности и, в конечном итоге, осуществить выбор метрического базиса для оценивания информационной безопасности программ.

Денотационный подход с использованием методов доказательного исследования и построения адекватных моделей программ позволяет анализировать правильность вычислительных структур, что позволит обнаружить ряд дефектов, НДВ и программных закладок при статическом анализе программы с использованием метрик сложности. Этот анализ осуществляется на основании языка структур, свойственного любой программе.

Аксиоматическая трактовка дает возможность установления соответствия (верификации) свойств вычислимости исследуемой и «эталонной» программ.

Таким образом, дальнейшие исследования будут направлены:

- на структурирование и разработку моделей представления программы и требований к контролю отсутствия НДВ программного обеспечения, и, на этой основе, структуризации метрик сложности и выбору метрического базиса;
- на определение (на основании анализа статистических данных и данных, полученных в результате проведения экспериментов) «доверенных» значений метрик сложности, формирующих метрический «эталон» для статического анализа программы, в частности, «эталонных» характеристик, отражающих качество разработки и стиль программирования;

- на сопоставление вычисленных метрик сложности для отдельных участков кода и функциональных объектов программы с «доверенными» характеристиками на основе обоснованных критериев (решающего правила) с целью обнаружения искажений вычислительных структур и нарушений свойств вычислимости, и в конечном итоге, выявления НДВ программы.

## 2.2 Выбор моделей представления программ на основе структурированных семантик

Выбор моделей представления программ для выявления в них дефектов и НДВ будем осуществлять в соответствии с рассмотренными трактовками семантики программ.

С позиций структурированного представления семантик вычислений в дальнейшем будем использовать следующий базис модели вычислений: РАЗРЕШЕНИЕ, представленное в денотационном подходе, и ВЫБОР, представленный в аксиоматическом подходе. Этот базис, соответственно, отражает вычислительные структуры и свойства вычислимости процесса вычислений, что является достаточным для реализации статического метрического анализа и выявления дефектов и НДВ программ.

При использовании языка структур, свойственного любой программе, будем использовать полный базис модели вычислений: РАЗРЕШЕНИЕ, ВЫБОР и ПРЕДПИСАНИЕ [18].

Основой для разработки этих двух формальных моделей и проводимого в дальнейшем статического и метрического анализа программы с целью выявления дефектов и НДВ, является ее управляющий граф  $\Gamma (B, D)$ , где  $B = \{b_i\}$  – множество вершин графа (линейных участков программы), а  $D \subseteq \{B \times B\}$  – множество дуг – переходов (связей по управлению между линейными участками).

Структурированные модели представления программы, основанные на анализе и упорядочении семантики вычислений, приведены на рисунке 2.1. Будем различать и использовать:

- формальную модель языка вычислительных структур, задаваемую грамматикой;
- формальную модель логических свойств вычислений, задаваемую продукционными системами.

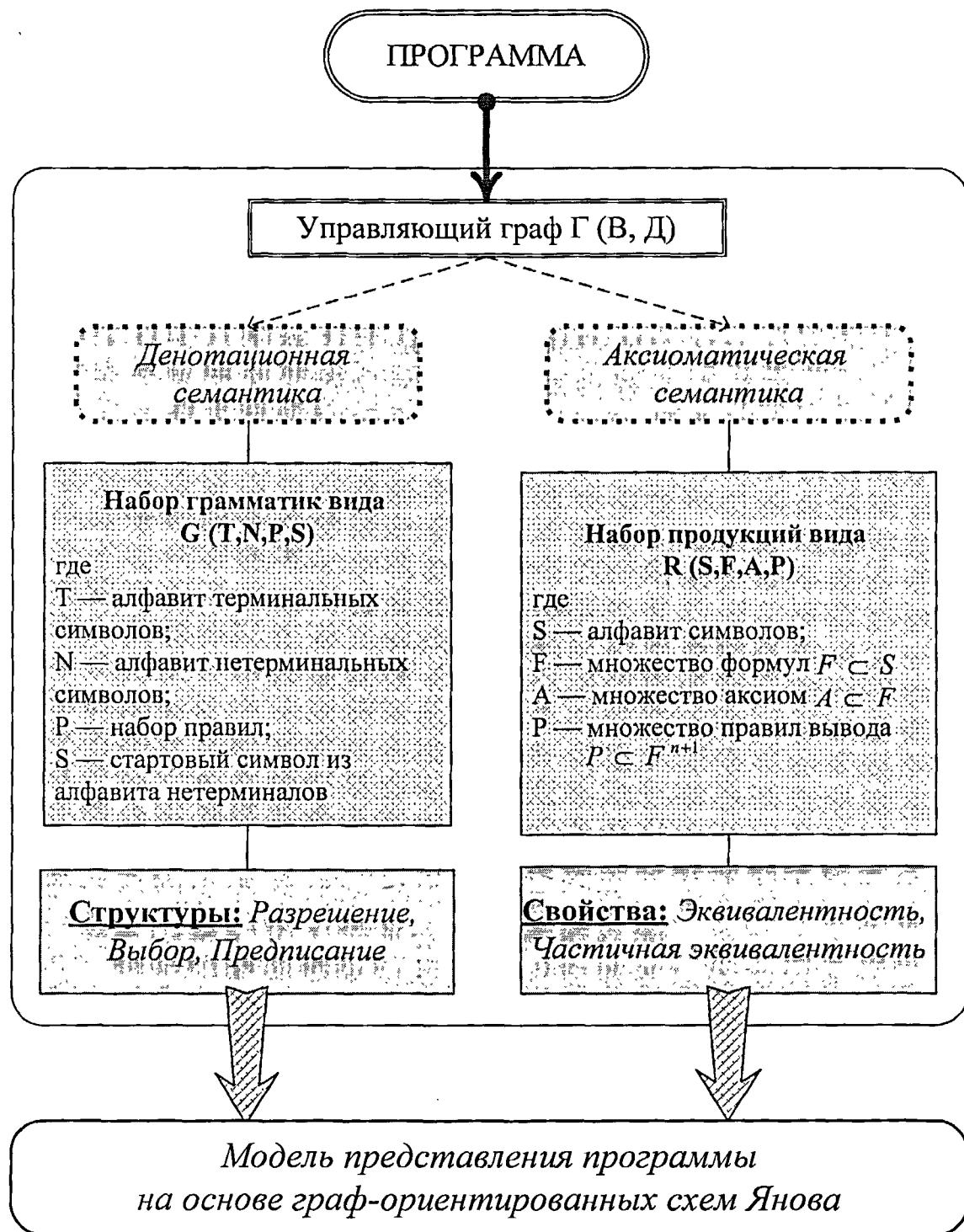


Рисунок 2.1 – Структурированные формальные модели представления программы

Каждая из предлагаемых двух моделей позволяет контролировать наличие/отсутствие соответствующих дефектов, в том числе НДВ программ за счет выявления неправильных вычислительных структур и некорректных свойств вычислимости. Реализация такого контроля

основывается на исследовании (вычислении) присущих каждой их моделей специфических характеристик.

Модель процесса вычислений, представленная в виде формальной грамматики, необходима для анализа правильности структур, которые в свою очередь, как указывалось выше, состоят из конструкций РАЗРЕШЕНИЕ, ВЫБОР и ПРЕДПИСАНИЕ. Грамматика однозначно определяет структуру каждой цепочки языка программирования, а также всей программы в целом.

В соответствии с классификацией, введенной Н. Хомским [7, 52], в качестве модели программы и процесса вычислений на основе формальных грамматик будем использовать контекстно-свободную и регулярную грамматики. Контекстно-свободная грамматика (КСГ) используется для описания структуры всего процесса вычислений в целом [79], а регулярная грамматика (РГ) будет использоваться для проверки правильности структур простейших конструкций (лексем), таких как идентификаторы, строки, константы и т. п.

Введем КСГ следующего вида:

$$G = (T, N, P, S), \quad (2.1)$$

где

$T = \{identifier, constant, \dots, register\}$  – набор терминальных символов языка ассемблер;

$N = \{ConditionalTransition, Moving, \dots, Shift\}$  – набор нетерминальных символов (табл. 2.1);

$R = \{ConditionalTransitionCommand, MovingCommand, \dots, IntoStackCommand\}$  – множество правил вывода;

$S \in T$  – стартовый символ.

КСГ имеет следующую специфику:  $n \rightarrow \beta$ , где  $\beta \in T \cup N$  и  $n \in N$ , то есть грамматика допускает появление в левой части правила только нетерминального символа. Это позволяет подняться над ограничениями

регулярных выражений и охватить все множество языков программирования.

Терминальные символы включают в себя все лексемы языка ассемблер, в том числе идентификаторы, константы, операторы, адреса памяти и т.д., то есть те символьные объекты, из которых строится, в частности, исходный текст ассемблерной программы. Разбор терминальных символов происходит на уровне регулярных грамматик, которые приведены ниже. Набор нетерминальных символов представляет собой множества лексем, объединенных по обобщающему признаку, а также их комбинаций, с использованием продукции. Примеры нетерминальных символов приведены в таблице 2.1.

Таблица 2.1 Множества терминальных и нетерминальных символов

Нетерминальный символ из $N$	Обобщающий признак	Терминальные символы из $T$
<i>ConditionalTransition</i>	Команды условного перехода	$JE   JNE   JA   JAE   JB   ...   JNA$
<i>Moving</i>	Команды переноса	$MOV   MOVS   MOVS8   ...   XCHG$
<i>Cycle</i>	Команды управления циклом	$LOOP   LOOPE   LOOPZ   LOOPNE   LOOPNZ$
<i>OperandFromStack</i>	Команды извлечения operandов из стека	$POP   POPA   POPAD   POPF   POPFD$
<i>OperandIntoStack</i>	Команды вставки operandов в стек	$PUSH   PUSHAD   PUSHF   PUSHFD$
<i>Shift</i>	Команды сдвига	$SHL   SHLD   SHR   SHRD$

Язык введенной грамматики  $G$  обозначим  $L(G)$  и назовем его языком структур. Язык структур представляет собой множество терминальных цепочек, порождаемых из стартового символа.

$$L(G) = \{w \in N \mid S \xrightarrow[G]{} w\} \quad (2.2)$$

В соответствии с определенной ранее спецификой, каждому выводу в КС-грамматике, начинающемуся с нетерминального символа, однозначно сопоставляется ориентированный граф, являющийся деревом и называемый деревом вывода (разбора) [8, 9, 34, 59]. Подчеркнем, что дерево вывода сопоставляется не грамматике, а конкретному выводу в данной грамматике, хотя необходимо заметить, что нескольким различным выводам может быть сопоставлено одно и то же дерево вывода.

Построение синтаксического дерева  $D_n$  по выводу  $S = w_0, w_1, \dots, w_n$  осуществляется следующим образом:

- 1) При  $m=0$   $D_0$  состоит из одной вершины, помеченной символом  $S$ , что является аксиомой грамматики  $G$ .
- 2) При  $0 \leq m \leq n$  для вывода  $S = w_0, w_1, \dots, w_n$  построено дерево  $D_m$  и  $w_m = x_1 A x_2$ ,  $w_{m+1} = x_1 \Psi x_2$ , и  $A \rightarrow \Psi$  – правило из  $R$ ,  $\Psi = x_1, x_2, \dots, x_r$ , где  $A$  – корневая вершина, а  $x_1, x_2, \dots, x_r$  ее потомки. В дереве  $D_m$  символы цепочки  $w_0, w_1, \dots, w_m$  являются метками его листьев, выписанных слева направо. Тогда дерево вывода  $D_{m+1}$  с цепочкой листьев  $w_0, w_1, \dots, w_m, w_{m+1}$  строится путем добавления новых листьев с метками  $x_1, x_2, \dots, x_r$  и соответствующих ребер.
- 3) Дерево  $D_n$  – искомое дерево для вывода  $S = w_0, w_1, \dots, w_n$ .

В случае полного вывода цепочка  $w_n$  состоит из терминальных символов, поэтому ни вывод, ни дальнейшие построения дерева вывода не возможны.

Следует отметить, что деревья вывода будут являться вершинами управляющего графа процесса вычислений, так как в вершинах графа будут находиться управляющие команды языка ассемблеров или строки, введенной грамматики.

Правильное построение дерева вывода ( $w_n \in T$ ) позволяет говорить об отсутствии нарушений в структуре процесса вычислений, и дает возможность отслеживать направление потока данных (регистр, память и стек). Перенаправление потока данных (смена источника и приемника), дает повод для подозрений на наличие вредоносных воздействий на процесс вычислений.

Отметим, что построение регулярной грамматики для процесса вычислений схоже с разработкой контекстно-свободной и регулярной грамматики [17]. Особенностью регулярной грамматики (РГ) является ее простота в сравнении с КСГ, что связано с ее специфическими свойствами правил вывода:

- $A \rightarrow B\gamma$  или  $A \rightarrow \gamma$ , где  $\gamma \in \Sigma$ ,  $A, B \in N$  (для леволинейных грамматик);
- $A \rightarrow \gamma B$  или  $A \rightarrow \gamma$ , где  $\gamma \in \Sigma$ ,  $A, B \in N$  (для праволинейных грамматик).

КСГ позволяют выявлять нарушения, связанные с общей структурой процесса вычислений, а РГ – нарушения, связанные с конкретными синтаксическими единицами (лексемами).

Модели процесса вычислений, представленные в виде формальной грамматики – контекстно-свободной и регулярной грамматики – позволяют проводить синтаксический и лексический анализ правильности структур программ, включая возможную их метрическую оценку.

Вывод о корректности свойств вычислимости можно сделать на основании понятий эквивалентности и частичности (частичной эквивалентности) и соответствующей выбранной модели представления программы.

Два алгоритма считаются эквивалентными, если они имеют одну и ту же область определения, реализуемые ими функции совпадают, а

система правил различна. Следовательно, понятие эквивалентности двух алгоритмов, для каждого из которых определено понятие «входа» и «выхода» (для каждого входа, который имеет смысл для данного алгоритма, выполнение алгоритма может приводить к некоторому выходу), можно ввести следующим образом. Алгоритмы считаются эквивалентными, если при условии равенства их «входов» будут равны и их «выходы» [5, 57].

На практике большее применение имеет общий способ введения эквивалентности, а именно определение эквивалентности с точностью до изоморфизма. В этом случае для выяснения вопроса об эквивалентности сопоставляются друг с другом не равные входы и выходы, а лишь находящиеся в соответствии, задаваемом изоморфизмом [5].

Между различными видами эквивалентности можно ввести частичное отношение порядка, выражющееся словами «сильнее» и «слабее» (частичная эквивалентность).

Очевидно, чем слабее отношение эквивалентности, тем шире классы алгоритмов, эквивалентных согласно этому отношению. Естественным является стремление расширить классы эквивалентных алгоритмов, вводя более слабое определение эквивалентности. Однако при слишком слабом определении эквивалентности возникают массовые проблемы, связанные с теорией алгоритмов, и среди них основная проблема распознавания эквивалентности алгоритмов, могут оказаться неразрешимыми. С другой стороны, слишком сильное определение эквивалентности чрезмерно сужает классы эквивалентных алгоритмов. Правильный выбор понятия эквивалентности играет большую роль как с точки зрения возможности получения содержательных теорем, так и с точки зрения их практической применимости.

На степень широты понятия эквивалентности наиболее существенное влияет выбор определения «выхода» алгоритма.

Классически под выходом понимаются объекты, которые формально объявляются результатами по окончании выполнения алгоритма. В некоторых случаях является важной информация, например, о промежуточных результатах или о том, в какой последовательности выполнялись действия алгоритма. Поэтому в самом общем смысле под выходом алгоритма следует понимать какую-то запись всей той информации, которую можно получить, наблюдая процесс выполнения алгоритма.

Таким образом, чем более общим является понятие выхода, тем более сильным оказывается отношение эквивалентности, основанное на таком определении выхода. Другими словами, все разнообразие эквивалентных алгоритмов состоит в том, что один и тот же конечный результат получается разными путями. Следовательно, чем больше истории о том, как выполнялся алгоритм, содержит в себе выход, тем к более сильному понятию эквивалентности он приводит. Это объясняется тем, что в один класс эквивалентных процессов попадают только те из них, история выполнения которых соответствует истории, зафиксированной в данном выходе.

Исчерпывающую информацию о том, как происходило выполнение алгоритма, можно получить, рассматривая в качестве выхода всю последовательность выполнявшихся операторов – значение операторного процесса вычислений. В этом случае процессы считаются эквивалентными, если их значения совпадают при одинаковых входах. Такое определение эквивалентности рассматривалось Ю.И. Яновым для операторных схем. Однако такое определение эквивалентности является слишком сильным, и многие процессы, которые разумно считать эквивалентными, при таком определении оказываются неэквивалентными [72].

Исходя из этого, в качестве выхода операторного процесса целесообразно рассматривать нечто такое, что по количеству информации о ходе выполнения процесса вычислений было бы чем-то средним между значением процесса и значением результативного переменного по окончании выполнения процесса вычислений. С этой точки зрения будем рассматривать в качестве выхода  $S$ -представления тех переменных, значения которых нас интересуют в качестве результатов выполнения операторного процесса вычислений [5].

Если ввести определение эквивалентности, базирующееся на использовании  $S$ -представления в качестве выхода, то тогда можно сказать, что два процесса эквивалентны по отношению к выделенным переменным в том случае, если соответствующие переменные при совпадающих входах вычисляются по одинаковым формулам.  $S$ -представление переменной есть не что иное, как явное выражение формулы, по которой вычислялось результирующее значение переменной для данных исходных значений функциональных переменных.

В программировании важную роль играют именно такие преобразования алгоритмов, которые оставляют расчетные формулы ( $S$ -представления) неизменными. Действительно, такие основные приемы программирования, как разбиение задачи на части, расчленение формул, выделение промежуточных результатов, преобразование логических операторов, экономия команд и ячеек памяти, приводят к таким преобразованиям алгоритмов, которые сохраняют  $S$ -представления результативных переменных.

В дальнейшем в качестве исходной модели для метрического анализа будет обоснована логическая модель представления программ, построенная с использованием граф-ориентированных схем Ю. И. Янова [71].

Общая безопасность программы и процесса вычислений зависит от выполнения отдельных функциональных объектов и отдельных участков кода (исходного, исполняемого или дизассемблированного кода). На практике при подтверждении и оценке соответствия исходные тексты программ зачастую (по различным причинам) отсутствуют, что предполагает проведение статического анализа дизассемблированного кода программ. В связи с этим оценка качества и безопасности программ на уровне дизассемблированного кода является наиболее универсальной, поскольку может проводиться как при наличии исходных текстов программ, так и в их отсутствие. Поэтому в дальнейшем будем также оперировать моделью программы, в которой рассматривается только дизассемблированный код, разбиваемый на различные сочетания базовых алгоритмических конструкций, что позволит контролировать изменения маршрутов выполнения в управляющем графе программы.

Таким образом, фундаментальными характеристиками программы, от которых в значительной мере зависит ее безопасность, являются структура и свойства программы.

При структуризации метрик сложности и формировании метрического базиса для оценивания безопасности программ необходимо связать модель программы, выбранную на основе анализа и упорядочения семантик, и совокупность метрик, характеризующих вычислительную структуру контролируемых программ и ее свойства вычислимости.

В плане вычислительной структуры и свойств вычислимости программы можно ограничиться рассмотрением наиболее чувствительных (к дефектам и преднамеренно вводимым искажениям программного кода) метрик топологической и информационной сложности, в основе которых лежат топологические характеристики граф-модели программы, а также структуры и модели данных. Метрики сложности этих двух групп

удовлетворяют подавляющему большинству требований, предъявляемых к метрическим показателям, а именно следующим требованиям:

- общность применимости;
- адекватность рассматриваемому свойству;
- существенность оценки;
- состоятельность;
- количественное выражение;
- воспроизводимость измерений;
- малая трудоёмкость вычислений;
- возможность автоматизации оценивания.

Совокупность метрик топологической и информационной сложности, наиболее чувствительных (по своему определению) к структурным искажениям и нарушениям свойств программы и наиболее значимых с позиций построения метрического базиса программы, приведены в модели структурированного представления программы (рис.2.2).



Рисунок 2.2. Соответствие логических моделей программы и метрик сложности

## 2.3 Построение логической модели программы с использованием граф-ориентированных схем Янова

Исходными данными для создания логической модели является управляющий граф программы. Классически управляющим графом программы считается ориентированный граф, содержащий лишь один вход и один выход, при этом вершины графа соотносят с теми участками кода программы, в которых имеются лишь последовательные вычисления, и отсутствуют операторы ветвления и цикла, а дуги соотносят с переходами от блока к блоку и ветвями выполнения программы. При этом каждая вершина достижима из начальной, и конечная вершина достижима из любой другой вершины [80].

В таком виде граф не пригоден для дальнейших исследований, поэтому преобразуем этот граф в форму операторной логической схемы Ляпунова [48], дающей возможность использования математического аппарата для операций с этими схемами. Для наиболее полной реализации возможности проведения исследований свойств эквивалентности и завершаемости процесса вычислений, позволяющих провести канонизацию и избавиться от избыточности, в качестве окончательной формы представления управляющего графа программы выберем логические схемы Янова [72, 28, 29], являющиеся частным случаем операторных схем и решающих задачу построения алгоритма для установления эквивалентности схем и нахождения полной системы эквивалентных преобразований. В итоге операторная схема Янова является моделью программы и реализуемого ею процесса вычислений, позволяющая исследовать корректность свойств вычислимости.

В общем случае алгебра схем Янова состоит из предикатных символов множества  $P\{p_1, p_2, \dots, p_k\}$ , операторных символов множества  $A\{A_1, A_2, \dots, A_n\}$  и графа переходов. Оператором в данной алгебре есть пара  $A_i\{P\}$ , состоящая из символа множества  $A_i$  и множества

предикатных символов. Графом переходов или операторной схемой Янова называется конечный ориентированный граф, обладающий следующими свойствами [29]. Вершины графа имеют не более двух выходящих из них дуг, представляемых стрелками. Вершины, из которых выходят по две стрелки, называются распознавателями, остальные называются операторами (преобразователями). К одной из вершин ведет специальная входная стрелка (рис. 2.3).

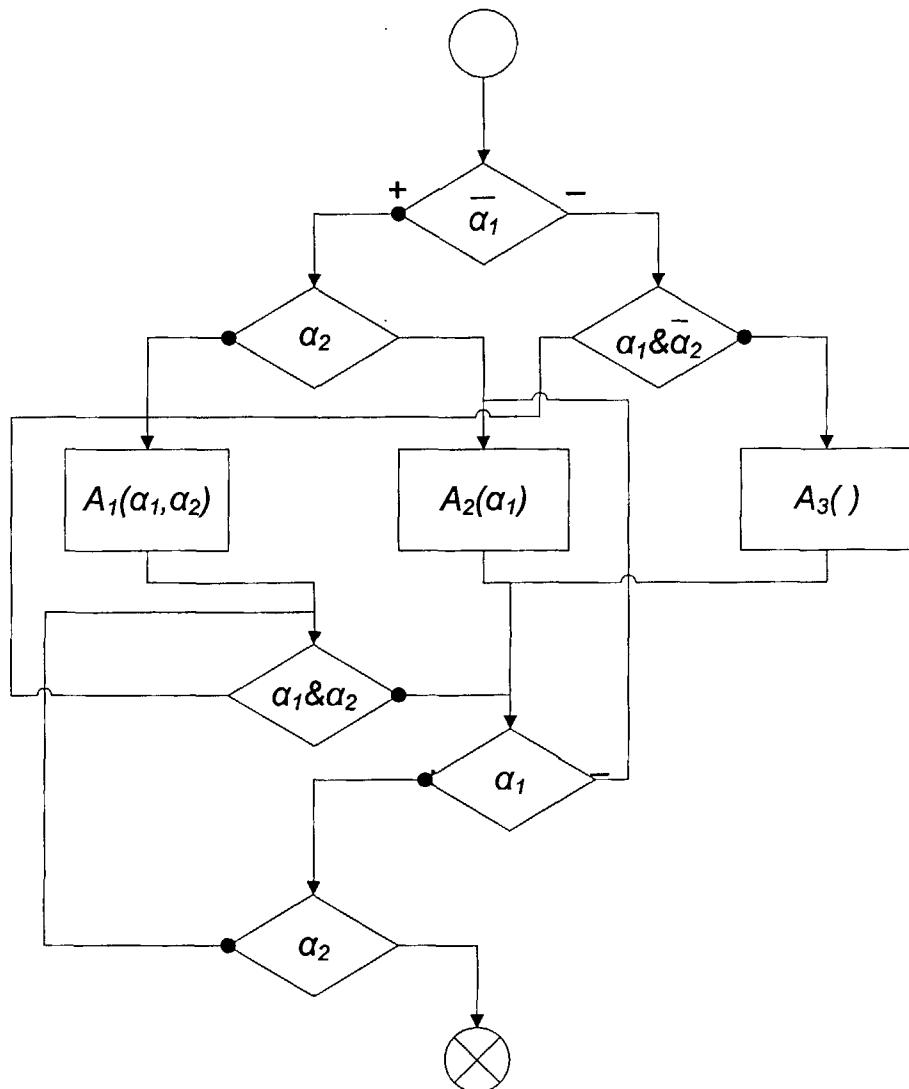


Рисунок 2.3 – Представление операторной схемы Янова

Стрелки, являющиеся выходами из распознавателя при истинности задаваемого им условия, называются плюс-стрелками, а, соответственно,

при ложности – минус-стрелками. В графе имеется только одна вершина, не имеющая выходной стрелки, которая называется конечным оператором.

Каждому распознавателю сопоставляется логическое условие выполнения схемы  $\alpha$  – произвольная функция алгебры-логики, зависящая от  $k$  логических переменных  $p_1, \dots, p_k$ . Распознаватель  $R$  с сопоставленным ему условием  $\alpha$  обозначается  $R(\alpha)$ .

Каждому оператору  $A$ , кроме конечного, приписывается некоторый набор  $P \subseteq \{p_1, \dots, p_k\}$  логических переменных, называемый сдвигом по логическим переменным. Оператор  $A$  с приписанным ему сдвигом обозначается  $A(P)$ .

Совокупность сдвигов  $P_i, \dots, P_n$  всех операторов  $A_i, \dots, A_n$  операторной схемы Янова называется распределением сдвигов, приписанным операторной схеме  $G$ . Все операторы одной операторной схемы считаются попарно различными. Операторная схема Янова  $G$  с множеством операторов  $A_i, \dots, A_n$ , множеством логических переменных  $p_1, \dots, p_k$  обозначается как  $G(A_i, \dots, A_n, p_1, \dots, p_k)$ .

В качестве конфигурации схемы рассматривается последовательность выполняемых операторов при прохождении по схеме от начала до конца. Конфигурация схемы  $G$ , строящаяся с помощью порождающего процесса по схеме  $G$ , задаваемого по индукции, будет получаться в виде двойной последовательности наборов значений логических переменных  $p_1, \dots, p_k$  и символов операторов  $A_i, \dots, A_n$ , записываемых в виде двух строк: верхней для наборов и нижней для операторов.

Построение конфигурации делается по индукции. В качестве базиса индукции в верхнюю строку записывается начальный произвольный набор  $\Delta_1 = \{p_1, \dots, p_k\}$ , нижняя строка пуста. Управление передается на начало схемы. При движении по схеме  $G$  возможны следующие варианты. Если

стрелка приводит к распознавателю  $R(\alpha)$ , то вычисляется  $\alpha(\Delta)$ . Если  $\alpha(\Delta) = 1$ , то с тем же набором движение продолжается по плюс-стрелке, выходящей из распознавателя  $R$ . Если  $\alpha(\Delta) = 0$ , то движение происходит по минус-стрелке. Если стрелка приводит к оператору  $A(P)$ , то выписываем символ оператора  $A$  в нижнюю строку конфигурации и преобразуем набор  $\Delta$  в набор  $\Delta'$ . При этом набор  $\Delta'$  отличается от  $\Delta$  не более чем значениями тех переменных из  $p_1, \dots p_k$ , которые входят в сдвиг  $P$  оператора  $A$ . Набор  $\Delta'$  выписывается в верхнюю строку конфигурации, и движение продолжается по стрелке, выходящей из  $A(P)$ . Процесс построения реализации обрывается при выходе на конец схемы или при бесконечном «зацикливании» на одних логических условиях, т.е. в случае, когда, не попав ни на один из операторов, движение приходит снова к недавно пройденному распознавателю. В этом случае движение искусственно прерывается и в нижнюю строку ставится символ пустого периода  $(\cdot)$ .

Для представленной на рис. 2.3 схемы конфигурации на множествах входных значений предикатов  $\{p_1 = 0, p_2 = 1\}$  и  $\{p_1 = 1, p_2 = 0\}$  имеет следующий вид:

- 1)    01            10            01  
 $A_1(0,1)$      $A_1(0,1)$     ...
- 2)    10            10  
 $A_3(\cdot)$     #

Две схемы  $G_1$  и  $G_2$  считаются эквивалентными ( $G_1 \cong G_2$ ), если совпадают их множества конфигураций, т.е.  $K(G_1) = K(G_2)$ . Эквивалентность схем обладает свойствами рефлексивности ( $G_1 \cong G_1$ ) и симметричности (если  $G_1 \cong G_2$ , то  $G_2 \cong G_1$ ).

Ю.И. Яновым был найден алгоритм распознавания эквивалентности любых двух схем и построена полная схема преобразований, содержащая 14 аксиом и три правила вывода [72]. Аксиомы в этом исчислении постулируют безусловную возможность замены некоторого фрагмента

операторной схемы на другой фрагмент, предписываемый данной аксиомой. Правила вывода постулируют возможность замены одного фрагмента на другой при выполнении некоторых условий, содержащихся в посылке данного правила вывода.

А.П. Ершов перенес теорию логических схем Янова на язык графов, в результате чего была упрощена аксиоматика преобразований: 14 аксиомам Янова соответствовало всего семь аксиом и три правила [29]. Кроме того, новый аппарат позволил эффективизировать правило вывода, использующее проверку логической подчиненности, которая, по Янову, производилась по сложному алгоритму преобразования схемы средствами, лежащими вне аксиоматики. Вместо этого алгоритма Ершовым было введено четыре аксиомы, которые задают обратимое преобразование схемы Янова, состоящее в разметке дуг управляющего графа булевскими функциями.

Н.А. Криницкий расширил систему преобразований Янова, введя в рассмотрение информационные связи между операторами [44, 57]. Оператор рассматривается как система функций, берущих свои аргументы из некоторых входных величин и записывающих результаты, в выходные величины оператора. Криницким была фактически рассмотрена функциональная эквивалентность (две схемы считаются эквивалентными, если соответствующие функции в любой интеграции одинаковы), найден алгоритм ее распознавания, построена каноническая форма и найдена полная система эквивалентных преобразований.

В дальнейшем основное внимание в развитие теории Ляпунова и алгебры Янова уделялось использованию схем программ для разработки эквивалентных преобразований не алгоритмов, а программ [55, 56, 69, 70].

Тем не менее, в рамках данного исследования аппарат алгебры Янова применяется к анализу именно алгоритмов. Представление алгоритмов программ в виде логических схем Янова позволяет явным образом

выявлять различные дефекты программы и НДВ, выражющиеся наличием тупиковых операторов, зацикливанием, наличием «мертвого» кода, избыточностью операторных узлов выбора, нарушением вложенности, разветвленности, протяженности или других свойств принятой программной структуры. Строгая формализация эквивалентных преобразований логических структур и распознавания эквивалентности алгебры Янова служит механизмом доказательства структурной корректности программных модулей и облегчает алгоритмизацию и автоматизацию универсальных процедур такого исследования. Непосредственно алгоритмы, используемые в них аксиомы и правила вывода для канонизации логической схемы Янова, представляющей исследуемое ПО, будут приведены в главе 3.

## **2.4 Построение метрической модели на основе требований к контролю отсутствия недекларированных возможностей**

Руководящий документ Гостехкомиссии (ФСТЭК) России «Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей» (РД НДВ) устанавливает классификацию программного обеспечения по уровню контроля отсутствия в нем НДВ и соответствующую данной классификации систему требований. В рамках проведенного исследования была рассмотрена только часть требований, являющихся составляющими элементами статического анализа исходных текстов программ.

На основании выполненного в главе 1 и п. 2.2 анализа существующих метрик сложности была проведена систематизация метрик сложности относительно требований РД НДВ, составляющих статический анализ программ (табл. 2.2). В качестве основного определяющего

критерия для установления соответствия было выбрано смысловое содержание метрики.

Таблица 2.2. Систематизация метрик топологической и информационной сложности относительно требований РД НДВ.

Наименование требования	Метрики сложности
<i>Статический анализ исходных текстов программ</i>	
Контроль полноты и отсутствия избыточности исходных текстов	<p><u>Метрики размера программы</u></p> <p><math>n_1, n_2</math> – количество уникальных операторов и operandов программы;</p> <p><math>n = n_1 + n_2</math> – словарь программы;</p> <p><math>N_1, N_2</math> – общее число операторов и operandов в программе;</p> <p><math>N = N_1 + N_2</math> – длина программы;</p> <p><math>N^* = n_1 \log_2 n_1 + n_2 \log_2 n_2</math> – теоретическая длина программы.</p>
Контроль связей функциональных объектов по управлению	<p><u>Метрика Майерса</u></p> $[V(G), n(G)],$ <p>где <math>V(G)</math> – цикломатическая мера</p> <p><math>n(G)</math> – число условий в коде программы плюс единица</p> <p><u>Метрика Вудворда</u></p> <p>Узловая мера (число узлов передач управления)</p> <p><u>Метрика точек пересечения</u></p> <p>Количество точек пересечения дуг графа программы (дуг передачи управления)</p>
Контроль связей функциональных объектов по информации	<p><u>Метрика Чепина</u></p> $Q = P + 2M + 3C + 0,5T$ <p>где <math>P</math> – вводимые переменные для расчетов и для обеспечения вывода;</p> <p><math>M</math> – модифицируемые или создаваемые внутри программы переменные;</p> <p><math>C</math> – переменные, участвующие в управлении работой программного модуля (управляющие переменные);</p> <p><math>T</math> – не используемые в программе («паразитные») переменные.</p>
Контроль информационных	<u>Метрика спена</u>

объектов	$SP_{cp} = \frac{1}{n} \sum_{i=1}^n SP_i$ <p>Спен — это число утверждений, содержащих данный идентификатор, между его первым и последним появлением в тексте программы.</p>
Формирование перечня маршрутов выполнения функциональных объектов	<p style="text-align: center;"><u>Метрика Мак-Клара</u></p> <p>Мера сложности, основанная на числе возможных путей выполнения программы, числе управляющих конструкций и переменных.</p> <p>Выделяются три этапа вычисления данной метрики:</p> <ol style="list-style-type: none"> <li>1) для каждой управляющей переменной вычисляется значения её сложностной функции по формуле:</li> </ol> $C_i = (D_i * J_i)/n,$ <p>где <math>D_i</math> - величина, измеряющая сферу действия переменной;</p> <p style="margin-left: 40px;"><math>J_i</math> - мера сложности взаимодействия модулей через переменную;</p> <p style="margin-left: 40px;"><math>n</math> - число отдельных модулей в схеме разбиения.</p> <ol style="list-style-type: none"> <li>2) для всех модулей, входящих в сферу разбиения, определяется значение их сложностных функций по формуле</li> </ol> $M(P) = fp * X(P) + gp * Y(P)$ <p>где <math>fp</math> и <math>gp</math> - соответственно число модулей, непосредственно предшествующих и непосредственно следующих за модулем <math>P</math>;</p> <p style="margin-left: 40px;"><math>X(P)</math> - сложность обращения к модулю <math>P</math>;</p> <p style="margin-left: 40px;"><math>Y(P)</math> - сложность управления вызовом из модуля <math>P</math> других модулей.</p>

Для представленных в таблице составляющих статического анализа аппарат программометрии является дополнительным инструментом, используемым в тех ситуациях при проведении испытаний, когда необходимо проконтролировать полученный с помощью основных методов результат.

Таким образом, совокупностью метрик сложности, соотносимых с указанными в РД НДВ требованиями к ПС на этапе статического анализа,

является набор семи метрик топологической и информационной сложности.

## Выводы по главе 2

В главе проанализированы основные подходы к формальному описанию семантик языков программирования. Наиболее подходящими для решения поставленной научной задачи представляются денотационный и аксиоматический подходы. Эти подходы позволяют рассматривать концепцию упорядоченных семантик, что делает возможным обосновать ряд структурированных моделей представления программы и соответствующих им структурированных групп метрик сложности и, в конечном итоге, осуществить выбор метрического базиса для оценивания информационной безопасности программ. Денотационный подход с использованием методов доказательного исследования и построения адекватных моделей программ позволяет анализировать правильность вычислительных структур, что позволит выявлять ряд дефектов, НДВ и программных закладок при статическом анализе программы с использованием метрик сложности. Аксиоматическая трактовка дает возможность установления соответствия (верификации) свойств вычислимости исследуемой и «эталонной» программ.

Таким образом, для целей выявления дефектов и НДВ определены две базовые модели представления программ в соответствии с рассмотренными трактовками семантики программ:

- Формальная модель языка вычислительных структур, задаваемую грамматикой, которая однозначно определяет структуру каждой цепочки языка программирования, а также всей программы в целом. Такая модель позволяет проводить синтаксический и лексический анализ правильности структур программ, включая возможную их метрическую оценку.

- Формальная модель логических свойств вычислений, задаваемую продукционными системами, на основании которой можно сделать вывод о корректности свойств вычислимости.

В качестве исходной модели для метрического анализа обоснована логическая модель представления программ на основе управляющего графа, построенная с использованием граф-ориентированных схем Янова.

Для оценивания безопасности программ необходимо связать модель программы, выбранную на основе анализа и упорядочения семантик, и совокупность метрик, характеризующих вычислительную структуру контролируемых программ и ее свойства вычислимости.

### **3 МЕТОД ВЫЯВЛЕНИЯ НЕДЕКЛАРИРОВАННЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ СТРУКТУРИРОВАННЫХ МЕТРИК СЛОЖНОСТИ**

#### **3.1 Общая характеристика метода выявления недекларированных возможностей программ на основе структурированных метрик сложности**

Сущность метода состоит в проведении следующих последовательных процессов:

1. Формирование метрического базиса как совокупности структурированных метрик топологической и информационной сложности, соотнесенных с конструкциями языка ассемблер, и последующее их ранжирование путем экспертного оценивания.

2. Приведение логической модели программы к продуктивной структуре, построенной на основе канонического представления схем Янова.

3. Экспериментальное определение «чувствительности» и корреляции выбранных компонент метрического базиса к изменениям программного кода, и, на этой базе, разработка системы критериев оценивания безопасности функциональных объектов (участков программного кода).

Для универсализации разработанного метода в качестве исходных данных рассматривается дизассемблированный код программы. На практике при подтверждении соответствия исходные тексты программ могут отсутствовать, что предполагает проведение статического анализа дизассемблированного кода программ. В связи с этим оценка качества и безопасности программ на уровне дизассемблированного кода является наиболее универсальной, поскольку может проводиться как при наличии исходных текстов программ, так и в их отсутствие.

Для разработки данного метода было выполнено построение метрического базиса как структурированной совокупности метрик топологической и информационной сложности, необходимых для проведения исследований. «Доверенные» характеристики, полученные опытным путем, образуют так называемые эталоны структуры и свойств вычислимости программ и в целом метрический эталон процесса вычислений.

### **3.2 Построение метрического базиса как совокупности структурированных метрик топологической и информационной сложности**

Одним из направлений обеспечения целостности и выявления НДВ программных средств может выступать контроль и анализ метрического базиса, отражающего такие важнейшие характеристики как структура и свойства вычислимости программ [11, 50, 68]. В аспекте структурной сложности программы можно ограничиться мерами, основанными на топологических характеристиках граф-моделей программ, которые удовлетворяют подавляющему большинству требований, предъявляемых к метрическим показателям, рассмотренным ранее. Именно топологические и информационные меры сложности наиболее применимы в целях построения метрического базиса программы, учитывающего ее структурные модификации и изменения свойств вычислимости, в котором видится решение задачи выявления участков кода программ, имеющих аномальные значения показателей качества и безопасности.

Приведённые меры удовлетворяют подавляющему большинству требований, предъявляемых к показателям: общность применимости, адекватность рассматриваемому свойству, существенность оценки, состоятельность, количественное выражение, воспроизводимость

измерений, малая трудоемкость вычислений, возможность автоматизации оценивания.

Именно топологические меры сложности наиболее часто применяются в фазе исследований, формирующей решения по управлению производством (в процессах проектирования, разработки и испытаний) и составляют доступный и чувствительный эталон готовой продукции, контроль которого необходимо планово осуществлять в период ее эксплуатации.

Первой разработанной топологической мерой сложности является цикломатическая мера Мак-Кейба [80]. Т. Мак-Кейб выдвинул и реализовал возможность оценивания сложности программы по числу базисных путей (маршрутов) в ее управляющем графе, т.е. всех возможных путей, соединяющих непрерывно любым сложным образом вход графа с его выходом. В общем случае цикломатическое число  $l(G)$  для графа  $G$ , имеющего  $n$  вершин,  $m$  дуг и  $p$  компонентов связности, можно рассчитать по формуле:

$$l(G) = m - n + p \quad (3.1)$$

Имеет место теорема, что число базисных путей в связном графе равно его цикломатическому числу, увеличенному на единицу. Цикломатической сложностью программы с управляющим (т.е., при  $p=1$ ) графом  $G$  называется величина  $V(G)$ , равная

$$V(G) = l(G) + 1 = m - n + 2 \quad (3.2)$$

Основными достоинствами цикломатической меры считаются простота ее вычисления и повторяемость результата, наглядность и содержательность интерпретации. В качестве недостатков можно отметить: нечувствительность к размеру и изменению структуры программы, отсутствие корреляции со структурированностью программы, отсутствие различия между базовыми конструкциями «развилка» и «цикл» и чувствительности к вложенности циклов. Наличие таких недостатков

цикломатической меры привело к появлению различных ее модификаций. Например, Дж. Майерс предложил метрикой считать интервал  $[V(G), n(G)]$ , где  $V(G)$ - цикломатическая мера, а  $n(G)$ - число условий в коде программы плюс единица, при этом оператор  $DO$  считается как одно условие, а оператор  $CASE$  из  $k$  вариантов – как  $k-1$  условий [50]. У. Хансен выдвинул идею рассматривать в качестве меры сложности пару  $\{l(G), N\}$ , где  $N$  – число операторов программы. Одним из вариантов цикломатической меры является метрика  $M(G) = (V(G), C, Q)$ , где  $C$  - количество условий, необходимых для покрытия управляющего графа минимальным числом маршрутов, а  $Q$  - степень связности структуры графа программы и ее протяженность.

Топологические метрики сложности, которые традиционно относят к второй группе, базируются на оценке вложенности управляющих конструкций в тексте программы. В эту группу входят тестирующая и функциональная меры (учитывают уровень вложенности ПО), метрика Пивоварского (учитывает цикломатическую сложность и глубину вложенности), метрика Мак-Клура (учитывает сложность схемы разбиения ПО на модули с учетом вложенности модулей и их внутренней сложности) и другие.

Для вычисления функциональной метрики сложности Харрисона-Мейджела первоначально необходимо приписать каждой вершине графа значение своей собственной первичной сложности в соответствии с оператором, который она изображает. Эта начальная сложность вершины может вычисляться любым способом, включая использование мер Холстеда.

Далее необходимо разбить граф на сферы влияния предикатных вершин. Для каждой предикатной вершины выделяется подграф, порожденный вершинами, которые являются концами исходящих из нее дуг, а также вершинами, достижимыми из каждой такой вершины (нижняя

граница подграфа), и вершинами, лежащими на путях из предикатной вершины в какую-нибудь нижнюю границу. Этот подграф называется сферой влияния предикатной вершины. Приведенной сложностью предикатной вершины называется сумма начальных или приведенных сложностей вершин, входящих в ее сферу влияния, плюс первичная сложность самой предикатной вершины. Следовательно, функциональную меру сложности ПО можно вычислить по формуле

$$f_1 = \sum c_i \quad (3.3)$$

как сумму приведенных сложностей всех вершин управляющего графа. При этом положительный результат дает одновременное применение дополнительной функциональной меры - отношения числа вершин графа к функциональному числу:

$$f^* = \frac{N_{c_1}}{f_1} \quad (3.4)$$

Метрика Пивоварского была разработана с целью учесть при оценке программы различия как между последовательными и вложенными управляющими конструкциями, так и между структурированными и неструктурированными программами. Рассчитывается по формуле:

$$N(G) = V^*(G) + \sum_i P_i, \quad (3.5)$$

где  $V^*(G)$  - модифицированная цикломатическая сложность, вычисленная аналогично  $V(G)$  с соблюдением условия, что оператор CASE из  $n$  вариантов рассматривается как один логический оператор;

$P_i$  - глубина вложенности  $i$ -ой предикатной вершины.

Под глубиной вложенности понимается число всех сфер влияния предикатов, которые полностью содержаться в сфере рассматриваемой вершины или пересекаются с ней. Глубина вложенности предикатных вершин рассчитывается на основе числа сфер влияния. При проведении сравнительного анализа цикломатических и функциональной мер с

метрикой Пивоварского для различных управляющих графов возможно сделать вывод, что мера Пивоварского возрастает при переходе от последовательных программ к неструктурированным при нечувствительности других мер сложности данной группы.

Метрика Мак-Клура предназначена, в первую очередь, для оценивания сложности структурированных программ в процессе проектирования. Она применяется к иерархическим схемам разбиения программ на отдельные функциональные модули, в результате чего возможно выбрать схему построения программ с меньшей сложностью еще до конкретной реализации самой программы. В качестве меры рассматривается зависимость сложности программы от числа возможных путей ее выполнения, числа управляющих конструкций и числа переменных. В формульном выражении метрика Мак-Клура рассчитывается как

$$M(P) = fp * X(P) + gp * Y(P), \quad (3.6)$$

где  $fp$  и  $gp$  - соответственно число модулей, непосредственно предшествующих и следующих за модулем  $P$ ;

$X(P)$  - сложность обращения к модулю  $P$ ;

$Y(P)$  - сложность управления вызовом из модуля  $P$  других модулей.

Основной особенностью применения данной метрики является то, что она ориентирована на хорошо структурированные программы, составленные из иерархических модулей, задающих функциональную спецификацию и структуру управления. Также подразумевается, что в каждом модуле одна точка входа и одна точка выхода, модуль выполняет ровно одну функцию, а вызов модулей осуществляется в соответствии с иерархической системой управления, которая задаёт отношение вызова на множестве модулей программы.

Метрику точек пересечения, авторами которой являются М. Вудворд, М. Хенел и Д. Хидлей, ориентирована на анализ неструктурированных программ, написанных на таких языках, как, например, ассемблер и Фортран, т.к. в хорошо структурированной программе таких ситуаций возникать не должно. Поток управления программы считается более сложным, если возникают точки пересечения дуг передачи управления. В графе программы, где каждому оператору соответствует вершина, т. е. не исключены линейные участки, при передаче управления от вершины  $a$  к  $b$  номер оператора  $a$  равен  $\min(a, b)$ , а номер оператора  $b$  -  $\max(a, b)$ . Точка пересечения дуг появляется, если

$$\begin{aligned} \min(a, b) < \min(p, q) &< \max(a, b) \& \max(p, q) > \max(a, b) | \\ \min(a, b) < \max(p, q) &< \max(a, b) \& \min(p, q) < \min(a, b). \end{aligned} \quad (3.7)$$

То есть, точка пересечения дуг возникает в случае выхода управления за пределы пары вершин  $(a, b)$ . Количество точек пересечения дуг графа программы дает характеристику неструтурированности программы.

Тестирующей мерой или метрикой Пратта называется мера сложности, удовлетворяющая следующим условиям:

1. Мера сложности простого оператора равна 1

$$2. M(\{F_1, F_2, \dots, F_n\}) = \sum_{i=1}^n M(F_i) \quad (3.8)$$

$$3. M(IF P THEN F_1 ELSE F_2) == 2MAX(M(F_1), M(F_2))$$

$$4. M(WHILE P DO F) = 2M(F)$$

Значение данной метрики возрастает с увеличением глубины вложенности и учитывает протяженность программы [11]. К тестирующей мере близко примыкает мера на основе регулярных вложений. Идея этой меры сложности программ состоит в подсчете суммарного числа символов (операндов, операторов, скобок) в регулярном выражении с минимально необходимым числом скобок, описывающим управляющий граф

программы. Указанные чувствительны к вложенности управляющих конструкций и к протяженности программы. Но использование данных метрик затрудняет большая трудоемкость вычислений.

Рассмотрим третью группу метрик сложности, опирающихся на характер разветвлений в тексте программы. Наиболее простой из этой группы считается узловая мера Вудворда и Хедли, основанная на подсчете топологических характеристик потока управления, при этом под узловой сложностью понимается число узлов передач управления. Значение метрики равно числу пересечений линий передачи управления от одного оператора к другому. Данная мера чувствительна к сложности линеаризации программы и ее структуризации. Ограничением применения этой метрики является возможность сравнения только эквивалентных программ. Тем не менее, данная метрика с учетом ограничений все же предпочтительнее метрик Холстеда, но по универсальности уступает метрике Мак-Кейба.

Метрика Чена выражает сложность программы числом пересечений границ между областями, образуемыми блок–схемой программы. Она применима только к хорошо структурированным программам, в которых присутствует последовательное соединение управляющих конструкций. Для неструктурированных программ метрика Чена существенно зависит от наличия и вложенности условных и безусловных переходов, что приводит к ее интервальной оценке. В этом случае устанавливаются верхняя, равная  $m+1$ , где  $m$  - число логических операторов при их гнездовой вложенности, и нижняя, равная 2, границы значений метрики. Если управляющий граф программы имеет одну компоненту связности, то метрика Чена совпадает с цикломатической метрикой Мак-Кейба.

Метрики Джилба оценивают сложность программ отношением числа операторов определенного вида к общему числу исполняемых операторов [32]. Одной из наиболее простых, но, как показывает практика, достаточно

эффективных является метрика, в которой логическая сложность программы определяется как насыщенность программы выражениями типа *IF-THEN-ELSE*, т.е. операторами перехода по условию. При этом вводятся характеристики абсолютной и относительной сложности программы, характеризующиеся количеством операторов условия. Используя дополнительные характеристики, такие, как максимальный уровень вложенности и число межмодульных связей, можно успешно применять метрики Джилба к анализу циклических конструкций и модульности программы. Данные метрики применяются для использования при оценивании сложности эквивалентных схем программ, в особенности схем Янова.

Первоначально целесообразно проведение экспресс-исследования структурной правильности программы. Оно осуществляется сверткой управляющего графа программы, состоящего из конструкций базиса *{Последовательность, Развилка, Цикл}*, которые могут объединяться и вкладываться одна в другую так, что граф программы остаётся правильной конструкцией. Минимальный уровень топологической сложности при этом должен быть равен единице, а более высокий показатель сигнализирует о структурных несовершенствах программы. Названная процедура применяется в современных компиляторах при оптимизации программ.

Дальнейшее исследование структурной сложности программ связано с выбором и приложением топологических и информационных мер сложности, чувствительных к изменениям частных аспектов управляющей структуры. Это изменение показателей вложенности конструкций, их разветвленности и протяженности, нарушение структурной взаимокорреляции и другие.

Из всех используемых метрических показателей, описывающих инвариантный базис программ, наиболее пригодными для оценки дефектов и связанных с ними НДВ программ являются показатели (метрики)

структуры программы. Для анализа стиля программирования целесообразно использование показателей информационных свойств (метрик информационной сложности) программ, которым присущи определенные недостатки.

Это связано с тем, что в качестве исходных данных при подсчете метрик используется дизассемблированный код. Процесс дизассемблирования является трудоемким и характеризуется изменчивостью конечного дизассемблированного кода. Доказать его корректность возможно, но при этом содержимое может быть различно для одного и того же исполняемого кода. Это, в первую очередь, влияет на разное количество информационных единиц в дизассемблированном коде при идентичных этапах обработки исполняемого кода. Соответственно, диапазон варьирования значений информационных метрик будет достаточно широк, что уменьшает точность выявления нарушений.

Отмеченный недостаток в гораздо меньшей степени влияет на количественные показатели структуры программы, так как процесс дизассемблирования исполняемого кода, который является источником вероятностных и информационных помех, практически не влияет на структуру программы.

Базис конструкций охватывает все возможные типовые структуры, которые могут существовать в исполняемом коде программы. Наличие пересечений стандартных типовых конструкций («Цикл-Цикл» – «Вложение», «Разветвление-Разветвление» – «Ветвление», «Последовательность-Последовательность» – «Составление») наиболее ярко отражает ключевые точки управляющего графа процесса вычислений. Простейшая конструкция «Разветвление» является самой распространенной из всех, и, следовательно, наиболее весомо отражает структуру процесса вычислений.

В результате исследований проведена структуризация и предложен базис конструкций языка ассемблер, который включает в себя все возможные комбинации основных алгоритмических структур *{Последовательность; Разветвление; Цикл}*.

Для анализа качества и безопасности программ предлагается использовать сочетание топологических и информационных метрик сложности [22]. Таким образом, требуется построение метрического базиса  $M = \{M_1, M_2\}$ , где  $M_1, M_2$  – множества показателей, описывающих, соответственно, метрические базисы топологической и информационной сложности.

Анализ всех топологических метрик выявил набор метрик, которые могут быть спроектированы на структурированный базис конструкций языка ассемблер и позволяют создать метрический базис структуры программы  $M_1$ . Базис структуры процесса вычислений  $M_1 = \{m_{11}, m_{12}, \dots, m_{19}\}$  в виде метрик представлен в таблице 3.1, и представляет собой набор соответствующих показателей  $m_{1j}, j=1, \dots, 9$ .

Таблица 3.1. Метрический базис структуры программы

Показатель	Конструкции языка ассемблер	Метрика
$m_{11}$	Последовательность Последовательностей	<p><u>Метрика Джилба</u></p> <p>Логическая сложность программы определяется как насыщенность программы выражениями типа IF-THEN-ELSE, в расширенном варианте применима и к анализу циклических конструкций.</p> <p>1. Отношение числа связей между модулями к числу модулей</p> $f = N_{sv}^4 / L_{sub}$ <p>2. Отношение числа ненормальных выходов из множества операторов к общему числу операторов</p> $f^* = N_{sv}^* / (L_{loop} + L_{if} + L_{sub})$ <p>где <math>L_{loop}</math> - число операторов цикла;</p> <p><math>L_{if}</math> - число операторов условного перехода;</p> <p><math>L_{sub}</math> - число связей между подпрограммами.</p>
$m_{12}$	Последовательность Разветвлений	<p><u>Метрика Чена</u></p> <p>Выражает сложность программы числом пересечений границ между областями, образуемыми блок-схемой программы.</p> $M(G) = (n(G), N, Q_0)$ <p>где <math>n(G)</math> - цикломатическое число;</p> <p><math>N</math> - число операторов;</p> <p><math>Q_0</math> - число пересечений</p>
$m_{13}$	Последовательность Циклов	<p><u>Метрика Вудворда</u></p> <p>Узловая мера (число узлов передач управления) <math>Y(X)</math></p>

Показатель	Конструкции языка ассемблер	Метрика
$m_{14}$	Разветвление Последовательностей	<p><u>Метрика Пивоварского</u></p> $N(G) = V^*(G) + \sum_i P_i$ <p>где <math>V^*(G)</math> - модифицированная цикломатическая сложность, вычисленная с условием, что оператор <i>CASE</i> с <math>n</math> выходами рассматривается как один логический оператор;</p> <p><math>P_i</math> - глубина вложенности <math>i</math>-ой предикатной вершины</p>
$m_{15}$	Разветвление Разветвлений	<p><u>Метрика точек пересечения</u></p> <p>Количество точек пересечения дуг графа программы дает характеристику неструктурированности программы.</p> <p>Точка пересечения дуг появляется, если</p> $\min(a, b) < \min(p, q) < \max(a, b) \quad \& \quad \max(p, q) > \max(a, b) \mid$ $\min(a, b) < \max(p, q) < \max(a, b) \quad \& \quad \min(p, q) < \min(a, b)$ <p>где <math>a, b</math> – вершины графа.</p>
$m_{16}$	Разветвление Циклов	<p><u>Функциональная метрика Харрисона-Мейджела</u></p> $f_1 = \sum c_1$ <p>- сумма приведенных сложностей всех вершин управляющего графа;</p> $f^* = N c_1 / f_1$ <p>- отношение числа вершин графа к функциональному числу</p>
$m_{17}$	Цикл Последовательностей	<p><u>Метрика регулярных выражений</u></p> $P(G) = N + L + \sum k,$ <p>где <math>N</math> - число operandов;</p> <p><math>L</math> - число операторов;</p> <p><math>\sum k</math> - число скобок в регулярном выражении управляющего графа программы</p>

Показатель	Конструкции языка ассемблер	Метрика
$m_{18}$	Цикл Разветвлений	<p><u>Метрика Пратта</u></p> <p>Мера сложности, удовлетворяющая следующим условиям:</p> <ol style="list-style-type: none"> <li>1. Мера сложности простого оператора равна 1.</li> </ol> $M(\{F_1; F_2, \dots, F_n\}) = \sum^n M(F_i)$ <ol style="list-style-type: none"> <li>3. <math>M(IF\_P\_THEN\_F_1\_ELSE\_F_2) = 2 \cdot MAX(M(F_1), M(F_2))</math></li> <li>4. <math>M(WHILE\_P\_DO\_F) = 2M(F)</math></li> </ol>
$m_{19}$	Цикл Циклов	<p><u>Метрика Мак-Клура</u></p> $M(P) = fp * X(P) + gp * Y(P),$ <p>где <math>fp</math> и <math>gp</math> - соответственно число модулей, непосредственно предшествующих и следующих за модулем <math>P</math>;</p> <p><math>X(P)</math> - сложность обращения к модулю <math>P</math>;</p> <p><math>Y(P)</math> - сложность управления вызовом из модуля <math>P</math> других модулей</p>

В главе 2 были определены базовые с точки зрения обеспечения безопасности информации свойства вычислимости программы, выведенные на основе ее структурированного представления. В ходе проведенного исследования был сделан вывод о том, что наиболее достоверно и полно данные свойства программы можно определить, базируясь на системе метрик информационной сложности, входящих в класс словарных метрик, описанных в п. 1.4. Дополнительно в рассматриваемое множество метрик информационной сложности были включены определенные в п. 2.4 метрики, отражающие информационные внутрипрограммные связи.

Базовой мерой оценивания количества информации является мера информационной энтропии, предложенная К. Шенноном на основе статистической формулы энтропии [33]. Информационную двоичную энтропию для независимых случайных событий  $x$  можно оценить по формуле:

$$H(x) = - \sum_{i=1}^n p(i) \log_2 p(i) \quad (3.9)$$

где  $n$  – число возможных состояний системы;

$p(i)$  - вероятность (или относительная частота) перехода системы в  $i$ -е состояние, причем сумма всех  $p(i)$  равна 1.

Такой подход к оценке информации не учитывает таких важных свойств информации, как ее ценность и смысл, но при этом использование двоичной системы (битов) позволяет оценивать абсолютно любые события. В дальнейшем предлагались другие меры количества информации, которые учитывали бы ее ценность и смысл, но сразу терялась универсальность и объективность: для разных процессов критерии ценности и смысла информации различны и субъективны.

Главной положительной стороной формулы Шеннона является ее отвлеченность от семантических и качественных, индивидуальных свойств системы. Она учитывает различность, разновероятность состояний – формула имеет статистический характер (учитывает структуру сообщений), делающий эту формулу удобной для практических вычислений. Основной отрицательной стороной формулы Шеннона является то, что она не различает состояния (с одинаковой вероятностью достижения, например), не может оценивать состояния сложных и открытых систем, применима лишь для замкнутых систем, отвлекаясь от смысла информации. Теория Шеннона, по сути, была разработана как теория передачи данных по каналам связи, и мера Шеннона – это мера количества данных, и не отражает их семантического смысла.

Тем не менее, известно, что увеличение (уменьшение) значения метрики Шеннона свидетельствует об уменьшении (увеличении) энтропии (организованности) системы. Применяя метрику Шеннона для сравнения зафиксированной и существующей в настоящее время программных реализаций, можно оценить возможность их эквивалентности.

Меры, описывающие уровень реализации программы и ее интеллектуальное содержание являются метриками, входящими в систему метрических показателей Холстеда, описанных в п. 1.4.

Предположим, что существует некоторый язык программирования, называемый потенциальным, в котором все программы (по крайней мере, для некоторой предметной области) уже написаны и представлены в виде процедур или функций. Тогда для реализации любого алгоритма на этом языке потребуется всего два оператора (функция и присваивание) и  $\eta_2^*$  имен входных и выходных переменных. Поскольку в такой записи никакие слова не повторяются, то длина программы совпадает с ее объемом и равна

$$V^* = (\eta_2^* + 2) \log(\eta_2^* + 2) \quad (3.10)$$

Эта величина называется потенциальным (минимально возможным) объемом, а отношение

$$L = \frac{V^*}{V} \quad (3.11)$$

называется уровнем реализации программы. Данный метрический показатель характеризует степень компактности программы, экономичность использования изобразительных средств алгоритмического языка. Чем ближе значение  $L$  к единице, тем совершеннее программа.

Метрика интеллектуального содержания является инвариантной по отношению к языкам программирования реализации программы. Интеллектуальное содержание можно рассчитать по формуле:

$$I = \frac{V}{D} \quad (3.12)$$

С учетом (3.10) и того, что мера трудоемкости кодирования

$$D = \frac{\eta_1 N_2}{2\eta_2} \approx \frac{1}{L} \quad (3.13)$$

можно преобразовать выражение (3.9):

$$I = \frac{V}{D} \approx V \times L = \frac{V \times V^*}{V} = V^* \quad . \quad (3.14)$$

Эквивалентность интеллектуального содержания программы  $I$  и ее потенциального объема  $V^*$  свидетельствует о том, что рассматриваемая метрика является характеристикой информативности программы.

Этап кодирования при создании программы можно определить как запись разработанного алгоритма в терминах выбранного языка программирования, т.е. использование из словаря определенного языка соответствующей конструкции и ее смыслового наполнения. При формализации данного этапа в системе Холстеда кодирование программы есть  $N$ -кратная ( $N$  – длина программы) выборка операторов и операндов из словаря программы, включающего  $\eta$  слов. При этом в соответствие единичной операции выбора слова (оператора, операнда) ставится также и смысловая сложность данной операции. Таким образом, интеллектуальное содержание программы непосредственно зависит от объема программы и трудоемкости ее кодирования.

Данная метрика характеризует интеллектуальную целостность исследуемых процедур программы, в результате чего с помощью данной характеристики можно, по сути, оценить умственные затраты на создание программы.

Основное назначение метрики Чепина состоит в оценке информационной «прочности» внутри каждого программного модуля и/или функционального объекта с помощью анализа характера использования входных и выходных переменных [32].

Для получения значения метрики все входные и выходные переменные исследуемого модуля разбиваются на четыре группы с соответствии с их применением:

- 1)  $P$  – переменные, получаемые извне для данного модуля;
- 2)  $M$  – переменные, модифицируемые или создаваемые внутри модуля;
- 3)  $C$  – переменные, участвующие в управлении работой программного модуля (управляющие переменные);
- 4)  $T$  – не используемые (т.н. «паразитные») переменные.

Поскольку каждая переменная может выполнять одновременно несколько функций, необходимо учитывать ее в каждой соответствующей функциональной группе.

Метрика Чепина в общем случае вычисляется по формуле:

$$Q = a_1P + a_2M + a_3C + a_4T, \quad (3.15)$$

где  $a_1, a_2, a_3, a_4$  – весовые коэффициенты.

Весовые коэффициенты используются для отражения различного влияния на сложность программы переменных из каждой функциональной группы. По мнению автора метрики, наибольший вес, равный трем, имеет функциональная группа  $C$ , так как она влияет на поток управления программы; весовые коэффициенты для других групп определены как  $a_1=1$  и  $a_2=2$ . Весовой коэффициент группы  $T$  не равен нулю, а имеет значение  $a_4=0,5$ , поскольку неиспользуемые переменные практически не увеличивают сложность потока данных, но создают информационную избыточность программы. С учетом определенных таким образом весовых коэффициентов формула метрики Чепина имеет вид:

$$Q = P + 2M + 3C + 0,5T \quad (3.16)$$

Метрика спена основывается на локализации обращений к данным внутри каждой программной секции. Спен называется число утверждений, использующих переменную, между его первым и последним появлением в

тексте программы. Следовательно, переменная, появившаяся в программе  $n$  раз, имеет спен, равный  $n-1$ . В качестве значения метрики принимается среднее значение от спенов, рассчитанных по всем используемым в программе переменным. При большом значении метрики спена усложняется тестирование и отладка программы.

Совокупность метрик, которые отражают информационные свойства программ и формируют метрический базис информационной сложности программ – инвариант свойств процесса вычислений –  $M_2 = \{m_{21}, m_{22}, m_{23}\}$ , приведены в таблице 3.2 ( $m_{2i}$ ,  $i=1,2,3$  – численные показатели свойств).

Таблица 3.2. Метрический базис информационной сложности программ

Показатель	Свойство вычислимости программы	Метрика
$m_{21}$	Эквивалентность Частичная эквивалентность	<u>Метрики размера программы</u> $n_1, n_2$ – количество уникальных операторов и operandов программы; $n = n_1 + n_2$ – словарь программы; $N_1, N_2$ – общее число операторов и operandов в программе; $N = N_1 + N_2$ – длина программы; $N^* = n_1 \log_2 n_1 + n_2 \log_2 n_2$ – теоретическая длина программы.
$m_{22}$		<u>Информационная энтропия</u> $H(x) = -\sum_{i=1}^n p(i) \log_2 p(i)$ где $n$ – число состояний системы; $p(i)$ – вероятность перехода системы в $i$ -е состояние.
$m_{23}$		<u>Интеллектуальное содержание</u> $I = V \times L$ где $V$ – объем программы; $L = \frac{2 \cdot n_2}{n_1 \cdot N_2}$ – уровень реализации программы
$m_{24}$		<u>Метрика Чепина</u>

Показатель	Свойство вычислимости программы	Метрика
		$Q = P + 2M + 3C + 0,5T$ <p>где <math>P</math> – вводимые переменные для расчетов и для обеспечения вывода;</p> <p><math>M</math> – модифицируемые или создаваемые внутри программы переменные;</p> <p><math>C</math> – переменные, участвующие в управлении работой программного модуля (управляющие переменные);</p> <p><math>T</math> - не используемые в программе («паразитные») переменные.</p>
$m_{25}$		<p>Метрика спена</p> $SP_{cp} = \frac{1}{n} \sum_{i=1}^n SP_i$ <p>– среднее значение спенов</p>

Таким образом, полным метрическим пространством, «чувствующим» структурные изменения и вариации информационных свойств программы может выступать модель, построенная на основе предложенного метрического базиса.

В зависимости от целей и углубленности проводимого исследования программы может применяться определенная совокупность метрик сложности, составленная из элементов предложенного базиса с последующим ранжированием. При этом наряду с традиционными методами оценивания метрических показателей качества ПС, используются методы экспертного оценивания. Наиболее распространенными являются метод анализа иерархий (Т. Саати) и методы принятия решений и обработки нечисловой информации по Н.В. Хованову. Тем не менее, с учетом сложности в представлении неопределенности в суждениях экспертов и большого числа матриц попарных сравнений, наиболее оправданным для низкоуровневой оценки и

ранжирования полагается применение подхода на основе математической теории свидетельств (теории Демпстера-Шейфера).

Критериями экспертного оценивания при ранжировании метрик сложности, составляющих метрический базис, выступали правила комбинирования теории свидетельств. Основными элементами выступают начальные количественные весовые показатели метрик сложности – основные массы вероятности, определяющие субъективную степень уверенности; функция уверенности как общая степень уверенности и ее фокальные элементы с их базовыми вероятностями, полученными, в данном случае, на основе свидетельств по правилу Демпстера.

Это правило позволяет для каждой совокупности фокальных элементов на всем множестве исходных данных сформировать результирующие подмножества и вычислить для них степени уверенности (результирующие комбинированные массы вероятностей), т.е. получить обобщенные весовые показатели метрик сложности, составляющих метрический базис для оценивания безопасности программы [38].

Для оценивания и ранжирования метрик сложности были привлечены три эксперта ( $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ ), которым было предложено назначить весовые показатели метрикам, составляющим метрический базис (табл. 3.1, 3.2). Таким образом, рассматривалось множество из 14 альтернатив  $A = \{a_1, a_2, \dots, a_9, a_{10}, \dots, a_{14}\}$ , соответствующее метрическому базису  $M = \{m_{11}, m_{12}, \dots, m_{19}, m_{21}, \dots, m_{25}\}$ , которым экспертами присвоены следующие количественные весовые показатели (основные массы вероятности):

$\mathcal{E}_1: p_1\{a_1\}=0,15; p_1\{a_2, a_3, a_4, a_6\}=0,1; p_1\{a_5\}=0,15; p_1\{a_7, a_8\}=0,08;$   
 $p_1\{a_9\}=0,06; p_1\{a_{10}\}=0,18; p_1\{a_{12}\}=0,2; p_1\{a_{11}, a_{13}, a_{14}\}=0,08.$

$\mathcal{E}_2: p_2\{a_1\}=0,18; p_2\{a_2, a_3, a_4, a_6\}=0,06; p_2\{a_5\}=0,19; p_2\{a_7, a_8\}=0,02;$   
 $p_2\{a_9\}=0,03; p_2\{a_{10}\}=0,3; p_2\{a_{12}\}=0,2; p_2\{a_{11}, a_{13}, a_{14}\}=0,02.$

$\mathcal{E}_3: p_3\{a_1\}=0,2; p_3\{a_2, a_3, a_4, a_6\}=0,08; p_3\{a_5\}=0,2; p_3\{a_7, a_8\}=0,06;$   
 $p_3\{a_9\}=0,08; p_3\{a_{10}\}=0,15; p_3\{a_{12}\}=0,15; p_3\{a_{11}, a_{13}, a_{14}\}=0,08.$

Уже на этом этапе, исходя из веса присвоенных экспертами начальных масс вероятностей, возможно предварительное объединение элементов в группы со следующими весовыми показателями:

$$\mathcal{E}_1: p_1\{a_1, a_5, a_{10}, a_{12}\} = 0,68; p_1\{a_2, a_3, a_4, a_6, a_9\} = 0,16; p_1\{a_7, a_8, a_{11}, a_{13}, a_{14}\} = 0,16$$

$$\mathcal{E}_2: p_2\{a_1, a_5, a_{10}, a_{12}\} = 0,87; p_2\{a_2, a_3, a_4, a_6, a_9\} = 0,09; p_2\{a_7, a_8, a_{11}, a_{13}, a_{14}\} = 0,04$$

$$\mathcal{E}_3: p_3\{a_1, a_5, a_{10}, a_{12}\} = 0,7; p_3\{a_2, a_3, a_4, a_6, a_9\} = 0,16; p_3\{a_7, a_8, a_{11}, a_{13}, a_{14}\} = 0,14$$

Далее, для получения обобщенной оценки были скомбинированы начальные значения вероятностей для всех пар экспертов, определены непустые пересечения маргинальных фокальных элементов, рассчитаны степень конфликтности и константа нормализации, и проведена нормализация комбинированных масс вероятностей.

Полученные результирующие комбинированные основные назначения вероятностей экспертов  $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$  имеют следующие значения:

$$\{m_{11}, m_{15}, m_{21}, m_{23}\} - p_{123}\{a_1, a_5, a_{10}, a_{12}\} = 0,7441$$

$$\{m_{12}, m_{13}, m_{14}, m_{16}, m_{19}\} - p_{123}\{a_2, a_3, a_4, a_6, a_9\} = 0,1636$$

$$\{m_{17}, m_{18}, m_{22}, m_{24}, m_{25}\} - p_{123}\{a_7, a_8, a_{11}, a_{13}, a_{14}\} = 0,0923$$

На основе анализа указанных результатов получена следующая ранжировка метрик сложности из состава метрического базиса:

$$\{m_{11}, m_{15}, m_{21}, m_{23}\} \succ \{m_{12}, m_{13}, m_{14}, m_{16}, m_{19}\} \succ \{m_{17}, m_{18}, m_{22}, m_{24}, m_{25}\} \quad (3.17)$$

Таким образом, наиболее предпочтительными являются метрические характеристики длины текста, метрики Джилба и точек пересечения, а также метрика интеллектуального содержания.

При проведении эксперимента были исследованы 48 типовых и уникальных программ и рассчитаны их метрические характеристики, при этом, помимо абсолютных значений метрик, оценивались относительные отклонения по участкам кода, математическое ожидание и среднеквадратическое отклонение в рамках исследуемой программы, примеры которых представлены в таблице 3.3.

Таблица 3.3. Результаты расчета метрик сложности.

<b>Метрика</b>	<b>Словарь <i>n</i></b>	<b>Кол-во строк кода</b>	<b>Теоре- тическая длина <i>N</i>*</b>	<b>Длина <i>N</i></b>	<b>Объем <i>V</i></b>	<b>Уровень реализа- ции <i>L</i></b>	<b>Интелл. содержа- ние <i>I</i></b>	<b>Метрика Джилба <i>f</i>*</b>	<b>Метрика точек пересе- чения</b>	
<b>Исследуемое ПС</b>										
<i>Исследование программного проекта с разделением по функциональным объектам</i>										
<b>Програм- мный проект № 24</b>	<b>ФО 1</b>	21	99	82	146	641,0	0,070	44,982	2,843	5586
	<b>ФО 2</b>	22	232	106	346	2561,0	0,021	54,784	2,825	5585
	<b>ФО 3</b>	17	138	69	250	526,7	0,118	61,962	2,772	5580
	<b>ФО 4</b>	21	239	112	336	3791,1	0,017	62,780	2,849	5592
	<b>ФО 5</b>	21	206	86	291	485,4	0,104	48,538	2,869	5582
	<b>ФО 6</b>	17	75	69	196	801,0	0,118	94,235	2,791	5593
	<b>ФО 7</b>	18	85	48	189	569,0	0,154	87,538	2,872	5589
	<b>ФО 8</b>	23	315	91	616	2786,0	0,025	69,651	2,828	5583
	<b>ФО 9</b>	19	81	28	96	304,0	0,222	67,556	2,779	5586
	<b>ФО 10</b>	16	78	38	114	394,0	0,182	71,636	2,793	5575
	<b>ФО 11</b>	15	122	28	96	304,0	0,222	67,556	2,894	5589
	<b>МО</b>	19,091		68,818	243,273	1196,664	0,114	66,474	2,831	5585,435
	<b>СКО</b>	2,663		29,872	153,340	0,076	0,076	14,859	0,041	5,017

Метрика Исследуемое ПС	Словарь <i>n</i>	Кол-во строк кода	Теорети- ческая длина <i>N</i> *	Длина <i>N</i>	Объем <i>V</i>	Уровень реализа- ции <i>L</i>	Интелл. содержа- ние <i>I</i>	Метрика Джилба <i>f</i> *	Метрика точек пересе- чения	Кол-во програм. закладок
<i>Исследование программных проектов с последующим внедрением в них типовых программных закладок</i>										
ПС №1	1	28	148	111	223	1072	0,04761	51,047	1,22	4412
	2	30	220	123	339	1663	0,04347	72,30	1,33	4603
ПС №2	1	18	74	71	107	446	0,07843	34,98	0,86	3143
	2	21	99	82	146	641	0,07017	44,98	1,02	3299
ПС №3	1	22	124	85	175	780	0,02105	16,42	1,15	5215
	2	26	232	106	346	1626	0,02139	34,78	2,09	5933
ПС №4	1	13	89	53	172	636	0,142857	90,86	2,14	4875
	2	17	138	69	250	1021	0,11764	120,12	2,60	5238
ПС №5	1	25	180	100	246	1137	0,01814	20,63	2,21	4978
	2	27	239	112	336	1597	0,01656	26,45	2,13	4924
ПС №6	1	9	189	33	224	710	0,2	142,00	1,08	3044
	2	13	305	48	406	1502	0,15384	231,08	1,27	3176
ПС №7	1	19	174	75	239	1015	0,1111	112,78	2,67	5567
	2	21	206	86	291	1278	0,1	127,80	2,89	5879
ПС №8	1	29	247	116	422	2050	0,02821	57,85	3,22	5812
	2	35	344	150	588	3016	0,02307	69,60	3,54	6083
ПС №9	1	26	268	101	636	2989	0,016216	48,47	2,87	5677
	2	31	445	147	763	3942	0,011	59,00	2,99	6384
ПС №10	1	11	66	43	248	857	0,1666	142,83	1,04	5489
	2	14	135	53	354	1347	0,14285	192,43	1,74	5614

Примечание. Номером «1» обозначен исходный вариант исследуемого программного проекта, номером «2» - вариант с внедренными в него типовыми программными закладками, количество которых указано в соответствующей строке.

Полученные данные использованы для формирования критериев распознавания признаков дефектов программ, подобных НДВ, и подготовки рекомендаций по их устранению. Примеры рассчитанных метрических показателей и корреляция их с имеющимися в программах дефектами представлены в таблице 3.4.

Таблица 3.4. Результаты расчета метрик сложности и корреляция их с имеющимися в программах дефектами.

Метрика Исслед ПС	Кол-во модуле й	Кол-во дефект ов	метрики Холстеда					Метри- ка Джилба $f^*$	Метри- ка точек пересе- чения
			Словарь $n$	Длина $N$	Теор. длина $N^*$	Уровень реализа- ции $L$	Интелл содерж ание $I$		
ПС №11	3	1	20	103	86,21	0,0027	87,11	1,02	2703
ПС №12	6	4	43	208	197,03	0,007	116,96	1,333	3874
ПС №13	9	6	21	174	70,71	0,0026	34,07	0,556	5462
ПС №14	21	18	27	287	128,38	0,0026	109,74	2,825	5542
ПС №15	43	36	35	445	165,93	0,0029	138,54	2,93	5617
ПС №16	41	39	15	330	149,78	0,0017	51,46	2,098	5501
ПС №17	49	44	21	620	92,24	0,0022	200,59	2,917	5621
ПС №18	74	57	24	323	199,33	0,0028	98,54	3,041	5947
ПС №19	74	76	28	475	226,36	0,0022	115,24	3,757	6166
ПС №20	76	89	30	553	236,48	0,0017	104,23	3,632	6327
Коэффициент корреляции	0,9452		0,9424	0,9897	0,9849	0,9387	0,9544	0,9746	0,9321

Проведенные экспериментальные исследования выбранных метрик сложности при наличии в программе искажений показали высокий уровень их корреляции с общим количеством дефектов (коэффициент корреляции колеблется в диапазоне 0,93 – 0,97), но недостаточную «чувствительность»

к конкретным дефектам – программным закладкам. Для выявления программных закладок требуется предварительно устраниТЬ большую часть дефектов, не относящихся к НДВ, что возможно при преобразовании управляющего графа программы в продуктивную структуру, построенную на основе канонического представления схем Янова.

### **3.3 Приведение программы к продуктивной структуре на основе канонического представления схем Янова**

Представление алгоритмов программ в виде логических схем Янова позволяет явным образом выявлять различные дефекты программы и НДВ, выражющиеся наличием тупиковых операторов, зацикливанием, наличием «мертвого» кода, избыточностью операторных узлов выбора, нарушением вложенности, разветвленности, протяженности или других свойств принятой программной структуры. Строгая формализация эквивалентных преобразований логических структур и распознавания эквивалентности алгебры Янова служит механизмом доказательства структурной корректности программных модулей и облегчает алгоритмизацию и автоматизацию универсальных процедур такого исследования.

Для проведения таких исследований необходимо определить универсальную модель представления программы с помощью схем Янова и приведение ее к каноническому виду путем эквивалентных преобразований с целью выявления НДВ и других дефектов программы [54].

Для построения модели программы в виде логической схемы Янова необходимо:

- построить управляющий граф программы  $\Gamma(B, D)$ , выступающий в роли начального отображения схемы Янова без предикатных условий;

- найти минимальное покрытие всех путей графа  $\Gamma(B, \Delta)$ , определяющее маршруты, минимально покрывающие соответствующую ему схему  $G(A, P)$ ;
- определить предикатные условия схемы  $G(A, P)$  путем символьического выполнения программы по найденным маршрутам;
- на основании управляющего графа  $\Gamma(B, \Delta)$  и найденных предикатных условий завершить построение схемы  $G(A, P)$ .

Для решения поставленной задачи представим управляющий граф программы в виде матрицы смежности.

Полной матрицей смежности графа  $\Gamma(B, \Delta)$ , где  $B = \{b_1, \dots, b_n\}$ , а дуга (ребро) понимается как пара вершин  $(b_i, b_j)$ , называется квадратная матрица  $A(\Gamma)$  порядка  $n$ , для которой

$$a_{ij} = \begin{cases} 1, & (b_i, b_j) \in \Delta \\ 0, & (b_i, b_j) \notin \Delta \end{cases} \quad (3.18)$$

Для нахождения минимального покрытия управляющего графа  $\Gamma(B, \Delta)$  воспользуемся методом Янова, описанным в [71] и включающим следующие общие процедуры:

- 1) приведение управляющего графа  $\Gamma(B, \Delta)$  к виду, имеющему одну точку входа и одну точку выхода –  $\Gamma_1(B, \Delta)$ ;
- 2) выделение в полученном графе  $\Gamma_1(B, \Delta)$  сильносвязанных подграфов (вершины которых взаимно достижимы) и построение нового ациклического графа  $\Gamma_2(B, \Delta)$ , каждая вершина которого соответствует сильносвязанному подграфу графа  $\Gamma_1(B, \Delta)$ , путем проведения эквивалентных преобразований [46];
- 3) построение минимального числа путей, покрывающих приведенный граф  $\Gamma_2(B, \Delta)$ ;
- 4) преобразование построенных путей для графа  $\Gamma_2(B, \Delta)$  в аналогичные пути для графа  $\Gamma_1(B, \Delta)$  (детализация сильносвязанных подграфов).

Наиболее трудоемкими и менее формализованными задачами являются приведение графа к ациклическому виду и нахождение минимального покрытия путей. Рассмотрим их подробнее.

Исходными данными для алгоритма выделения сильносвязанных подграфов и приведения графа  $\Gamma (B, D)$  к ациклическому виду является построенная ранее матрица смежности  $A(\Gamma)$  порядка  $n$ . Далее выполняются следующие эквивалентные преобразования:

Шаг 1. Вычисляется матрица первой достижимости по формуле

$$R_1(\Gamma) = A(\Gamma) + E, \quad (3.19)$$

где  $E$  – единичная матрица.

Шаг 2. Вычисляется матрица достижимости графа путем возвведения матрицы первой достижимости в степень в соответствии с правилами булевой алгебры. Данный шаг выполняется до получения значения степени, равного  $n$ :

$$R(\Gamma) = R_1^n(\Gamma), \quad (3.20)$$

где  $n$  – порядок матрицы смежности  $A(\Gamma)$ .

Шаг 3. По полученной матрице достижимости строится матрица связности:

$$S(\Gamma) = R(\Gamma) \& R^T(\Gamma), \quad (3.21)$$

где  $R^T(\Gamma)$  – транспонированная матрица достижимости,

$\&$  – операция поэлементного умножения матриц.

Шаг 4. Матрица связности  $S(\Gamma)$  приводится к блочно-диагональному виду путем перестановки местами ее строк и столбцов. В результате получаем матрицу  $S_1(\Gamma)$ , на главной диагонали которой находятся блоки, полностью состоящие из элементов, равных 2. Каждому такому блоку соответствует множество вершин графа, образующих (вместе с соответствующими дугами) сильносвязный подграф.

Шаг 5. Представляя каждый сильно-связанного подграф в виде одной вершины, получаем новый ациклический граф  $\Gamma_2 (B, D)$ , интегрально соответствующий исходному графу.

Далее рассмотрим алгоритм построения минимального числа путей, покрывающих полученный граф  $\Gamma_2 (B, D)$  для определения маршрутов, минимально покрывающих соответствующую ему схему  $G (A, P)$ . Исходными данными для этого алгоритма выступает являющаяся упакованная матрица смежности  $A' (\Gamma_2)$ . Использование упакованной матрицы смежности вместо полной обусловлено меньшим количеством столбцов в ней, следовательно, меньшим временем обработки, и более удобным представлением графа для моделирования движения по нему.

Упакованной матрицей смежности для графа  $\Gamma_2$  является матрица  $A' (\Gamma_2)$  размера ( $n' \times l$ ), где  $n'$  – количество вершин графа  $\Gamma_2$ ,  $l$  – максимальное суммарное количество входов и выходов из одной вершины. Элементы упакованной матрицы смежности определяются по правилу: номер строки матрицы соответствует номеру вершины графа, в строку с номером  $i$  записываются номера всех смежных вершин для вершины  $i$ . При этом каждой вершине графа (с номером  $i$ ) поставлена в соответствие пара чисел  $(v_{in}(i), v_{out}(i))$ , обозначающая количество входящих и выходящих дуг.

Шаг 1. Вершина  $i = 1$  является начальной.

Шаг 2. Если  $i$  – не конечная вершина графа, то определяем вершину  $j$  с максимальным номером, смежную с  $i$ .

Шаг 3. Если  $v_{out} (i) > 1$  и  $v_{in} (j) > 1$ , то дугу  $(i, j)$  исключаем из графа.

Шаг 4. Если  $v_{out} (i) = 1$  и  $v_{in} (j) > 1$ , а рассмотренная на предыдущей итерации дуга  $(j, k)$  не удалена из графа, удалить ее вместе с дугой  $(i, j)$ .

Шаг 5. Если  $v_{out} (i) > 1$  и  $v_{in} (j) = 1$ , то дугу  $(i, j)$  включаем в путь и помечаем.

Шаг 6. Если в путь входят отмеченные дуги, перейти к шагу 1 (построение следующего пути), иначе конец алгоритма.

Далее для формирования логической модели программы в виде схемы Янова происходит определение предикатных условий графа  $\Gamma_2(B, D)$  путем символьческого выполнения программы по полученным маршрутам. Алгоритм данного процесса представлен на рис. 3.2.



Рисунок 3.2 – Алгоритм определения предикатных условий графа  $\Gamma_2$   
 $(B, D)$ .

Построение схемы  $G(A, P)$  завершается формированием соответствующего базиса схем Янова и преобразованием графа в схему Янова на основании аксиом и правила вывода эквивалентных преобразований.

При формировании базиса схем Янова необходимо учитывать их ограничения, связанные с принадлежностью схем Янова к классу разрешимых схем.

Таким образом, в качестве базиса схем Янова  $\beta$ , определяемого для графа  $\Gamma_2(B, \Delta)$ , зададим множество, состоящее из одной переменной  $x$  и набора одноместных функциональных символов  $\{f_1, f_2, \dots, f_n\}$  и одноместных предикатных символов  $\{p_1, p_2, \dots, p_m\}$ :

$$\beta = ([\{x\}, \{f_1, f_2, \dots, f_n\}, \{p_1, p_2, \dots, p_m\}], \{x_i = f_i(x) | i = 1, 2, \dots, n\} \cup \{p_i(x) | i = 1, 2, \dots, m\}) \quad (3.22)$$

Заданный базис совместно с правилами преобразования управляющей структуры графа  $\Gamma_2(B, \Delta)$ , определенными на основе доказанных аксиом и правил вывода, описанных в [29], формируют соответствующую управляющему графу исследуемой программы  $\Gamma(B, \Delta)$  схему Янова  $G_\beta(A, P)$ .

Основной целью применения проводимых эквивалентных преобразований управляющего графа  $\Gamma(B, \Delta)$  является приведение его к эквивалентной канонической схеме, позволяющей выявить структурные дефекты исследуемой программы. Последующее выделение из канонической схемы продуктивной структуры программы дает возможность использования в дальнейшем при проведении испытаний минимально-представительного тестового покрытия программы, и создание условного модельного эталона программы, предотвращающего в дальнейшем ее модификацию.

Канонической операторной схемой Янова называется схема, все операторы которой выполнимы, и в которой все логические условия равны 1 только на допустимых наборах [29]. На рисунке 3.3 изображена

каноническая схема Янова, эквивалентная операторной схеме, представленной на рис. 2.3.

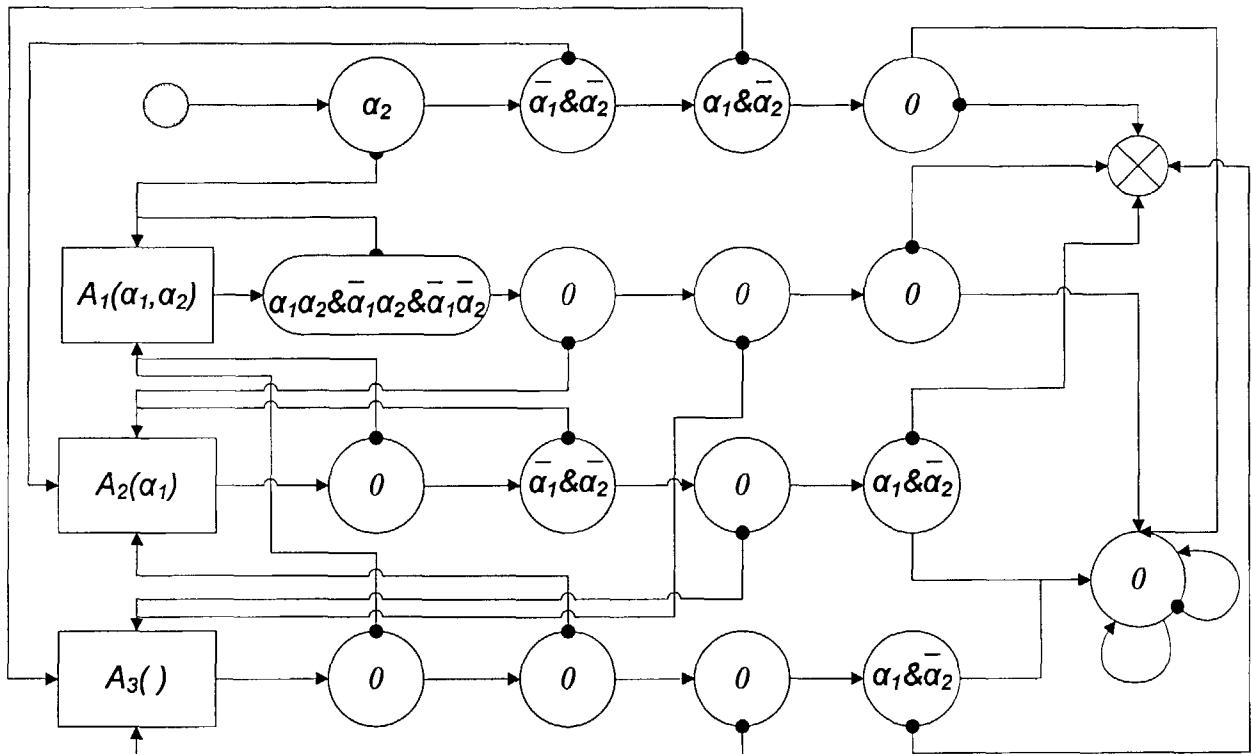


Рисунок 3.3 – Пример канонической схемы Янова

Алгоритм приведения построенной схемы Янова  $G(A, P)$  к каноническому виду, базирующийся на доказательстве теоремы о существовании для любой операторной схемы Янова равносильной ей канонической схемы и использующий аппарат алгебры Янова (аксиомы А, правила вывода П и технические теоремы ТТ), представленные в [29], изображен на рис. 3.4 и состоит из следующих этапов.

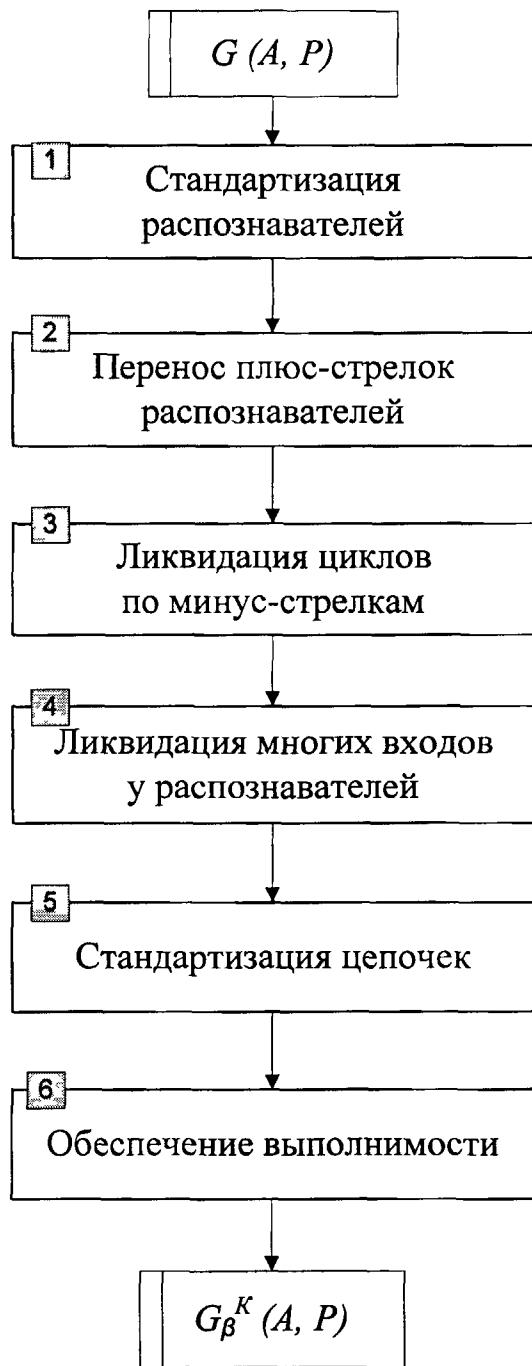


Рисунок 3.4 – Алгоритм приведения построенной схемы Янова  $G(A, P)$  к каноническому виду.

Шаг 1. Стандартизация распознавателей. По правилу П3 условие в каждом из нетождественно ложных распознавателей записывается в совершенной нормальной дизъюнктивной форме. Затем по аксиоме А3 все распознаватели расчленяются так, что условие в каждом распознавателе

имеет вид элементарной конъюнкции (любые элементарные конъюнкции  $\delta_1$  и  $\delta_2$  либо равны, либо ортогональны, т.е.  $\delta_1 \wedge \delta_2 = f$  ).

Шаг 2. Перенос плюс-стрелок. Процесс переноса плюс-стрелок для каждого из распознавателей  $R$  происходит по следующим правилам:

- а) если  $R$  – заглушка, стрелка не переносится;
- б) если  $R$  – полуцикл, т.е. распознаватель, у которого один из выходов является одновременно его входом, то применяется теорема ТТ7;
- в) если плюс-стрелка от  $R$  ведет к преобразователю, останову или заглушке, стрелка не переносится;
- г) если плюс-стрелка от  $R(\alpha)$  ведет к распознавателю  $R(\beta)$ , то применяется аксиома А5, если  $\alpha=\beta$ , и ТТ11, если  $\alpha \wedge \beta = f$ .

Эти правила последовательно применяются к каждому из распознавателей схемы  $G(A, P)$ . Если какой-то из распознавателей после переноса стрелок окажется без входа, то он уничтожается по теореме ТТ6. При переносе плюс-стрелки могут встретиться два случая:

- перенос стрелки обрывается естественным образом в соответствии с правилами а) и б).
- стрелка от  $R(\alpha)$  после последовательности переносов через распознаватели  $R_1, \dots, R_t, t \geq 1$  опять вернулась на вход  $R_1$ . Это означает, что распознаватели  $R_1, \dots, R_t$  образуют цикл. В этом случае рассмотрим распознаватель  $R_1(\beta)$ .

Если  $\alpha=\beta$ , то плюс-стрелка от  $R_1(\alpha)$  должна идти вдоль цикла. Сначала начнем переносить плюс-стрелку от  $R_1(\alpha)$ . Она пройдет ту же последовательность переносов, что и в начале стрелка от  $R(\alpha)$ , т.е. после серии переносов плюс-стрелка от  $R(\alpha)$  попадет на вход  $R_1(\alpha)$ . Таким образом,  $R_1(\alpha)$  станет полуциклом, который по теореме ТТ7 трансформируется в заглушку. Плюс-стрелка от  $R(\alpha)$  переносится на эту

заглушку, которая по аксиоме А6 раздваивается, после чего восстанавливается первоначальный вид распознавателя  $R_I(\alpha)$ .

Пусть  $\beta \wedge \alpha = f$ . Тогда  $\alpha \wedge \neg\beta = \alpha$  или  $\neg\beta = \alpha \vee \gamma$ , где  $\gamma$  – некоторая совершенная нормальная форма. Применим к  $R_I(\alpha)$  аксиому А2.

Плюс-стрелка от  $R_I(\neg\beta) = R_I(\alpha \vee \gamma)$  пойдет вдоль цикла  $R_1, \dots, R_t$ .

Применив к  $R(\alpha \vee \gamma)$  аксиому А3, получим предыдущий случай, после чего восстановим распознаватель  $R_I(\alpha \vee \gamma)$  к виду  $R_I(\beta)$ .

В итоге у каждого распознавателя, не являющегося заглушкой, плюс-стрелка направлена или в заглушку, или в преобразователь, или в останов.

Шаг 3. Ликвидация циклов по минус-стрелкам. Если после выполнения шага 2 в схеме остались циклы, то распознаватели  $R_1, \dots, R_t$ , образующие эти циклы, соединены друг с другом только минус-стрелками. Если  $t=1$ , то такой полуцикл ликвидируется по теореме ТТ8.

Рассмотрим сначала правильный цикл  $R_1, \dots, R_t$ , не имеющий входов извне. В этом случае к  $R_I(\alpha)$  применяется аксиома А2 и он расчленяется на элементарные конъюнкции по аксиоме А3. Все возникшие при этом плюс-стрелки будут вести к распознавателю  $R_2(\beta)$ , и по правилам шага 2 осуществим перенос плюс-стрелок. В результате  $R_2(\beta)$  окажется без входных дуг и может быть уничтожен по теореме ТТ6. Поскольку  $R_1, \dots, R_t$  – правильный цикл, то далее могут быть последовательно уничтожены все распознаватели, входящие в этот цикл, и распознаватели, возникшие при расчленении  $R_I(\alpha)$ .

Рассмотрим теперь любой цикл  $R_1, \dots, R_t$ , имеющий извне  $l$  входов. Для любого из входов такого цикла верно следующее условие:

- если вход идет от распознавателя, после выполнения шага 2 это может быть только минус-стрелка распознавателя  $R_I(\alpha)$ ;

- если вход идет от преобразователя, то по аксиоме А1 в инверсной форме можно между ним и циклом вставить распознаватель  $R(0)$ .

Применим к распознавателю  $R(\alpha)$  аксиому А2 и, по аксиоме А3, расчленим его на элементарные конъюнкции. Все возникшие при этом плюс-стрелки будут входить в тот же распознаватель  $R_i, 1 \leq i \leq t$ , в который ранее входила минус-стрелка от  $R(\alpha)$ . Выполним перенос всех плюс-стрелок с распознавателя  $R_i$  по правилам шага 2. По окончании переноса ни одна из плюс-стрелок не соединена ни с одним из распознавателей  $R_1, \dots, R_t$ . Следовательно, цикл  $R_1, \dots, R_t$  теперь имеет  $t-1$  вход. Повторяя описанный процесс  $t$  раз, получаем правильный цикл, который можно уничтожить по описанному ранее алгоритму.

Шаг 4. Ликвидация многих входов у распознавателей. Поскольку в схеме нет циклов, для любого распознавателя  $R$  существует цепочка, ведущая от  $R$  либо к преобразователю, либо к останову, либо к заглушке конечной длины. Длину максимальной такой цепочки назовем высотой распознавателя  $R - h(R)$ . Если между двумя распознавателями есть дуга, то  $h(R) > h(R')$ . Применим к распознавателям, имеющим более одного входа, с максимальным значением высоты  $h_{max}$ , теорему ТТ4. В результате получаем схему с максимальным значением высоты, равным  $h_{max}-1$ . Осуществляя последовательно этот процесс размножения распознавателей по теореме ТТ4, получим схему, в которой все распознаватели, отличные от заглушек, будут иметь только по одному входу. Ликвидация нескольких входов в заглушки выполняется по аксиоме А6.

Шаг 5. Стандартизация цепочек. Для выполнения данного этапа необходимо, чтобы число цепочек в схеме равнялось  $n+1$ , где  $n$  – число операторов. Для этого, если входная стрелка схемы или выходная дуга преобразователя непосредственно ведет к какому-либо преобразователю, применим к ней аксиому А1 в инверсном виде, причем плюс-стрелка вставленного распознавателя  $R(0)$  ведет на останов.

Далее необходимо, чтобы минус-стрелка последнего распознавателя каждой цепочки вела на заглушку. Если такая стрелка ведет к преобразователю, то по аксиоме А1 в инверсном виде поставим вместо нее распознаватель  $R(0)$  с плюс-стрелкой, ведущей на заглушку. Применяя аксиому А2, заменяем  $R(0)$  на  $R(1)$  с плюс-стрелкой, ведущей на оператор, и с минус-стрелкой, идущей на заглушку, и затем расчленяем  $R(1)$  по аксиоме А3 на элементарные конъюнкции.

Для обеспечения ортогональности на основании теоремы ТТ10 сгруппируем распознаватели вдоль каждой цепочки так, чтобы все распознаватели с одинаковыми элементарными конъюнкциями стояли рядом, и, применяя теоремы ТТ3 и ТТ6, уничтожим все такие распознаватели, кроме первого.

Затем обеспечим, чтобы в каждой из  $n+1$  цепочек присутствовала передача управления на каждый из  $n$  операторов и на блок останова. Для этого по аксиоме А1 вставим в любом месте цепочки распознаватель  $R(0)$  с плюс-стрелкой, ведущей на недостающий преобразователь. После этого по теореме ТТ10 упорядочим распознаватели вдоль цепочки следующим образом:  $A_1, \dots, A_n$ , останов, заглушки, после чего все распознаватели с плюс-стрелками, ведущими на заглушки, сгруппируются в конце цепочки. Сделаем все заглушки по теореме ТТ5 вполне одинаковыми с условием, равным 0, далее по аксиоме А6 соберем все заглушки в одну. В результате на концах цепочек могут появиться распознаватели, оба выхода которых ведут к одному оператору, и устраним такие распознаватели по теореме ТТ2. По аксиоме А3 в инверсной форме сгруппируем в каждой цепочке все элементарные конъюнкции, ведущие к одному и тому же оператору, в один распознаватель.

В результате проведенных преобразований получим матричную схему  $G_M$ , равносильную исходной схеме  $G$ .

Шаг 6. Обеспечение выполнимости. С помощью аксиом A7 - A10 построим стационарную верхнюю разметку схемы  $G_M$  и применим правило П1 к каждому из распознавателей. В построенной матричной схеме  $G_M$  для

каждого  $R(\alpha)$  имеет место  $F\left\{\hat{O}_{R(F)}\left(\hat{O} = \max F(P_1, \dots, P_k)\right)\right\}$ , выполняется, что условия всех распознавателей равны единице только на допустимых наборах. Применением аксиом в инверсном виде A7 – A10 построенная разметка удаляется со схемы.

После этого по аксиоме A1 уберем все распознаватели  $R(\emptyset)$ . В результате этого в схеме могут появиться операторы, не имеющие входов. Убрать такие операторы и выходящие из них цепочки распознавателей можно последовательным применением аксиомы A4 и теоремы ТТ6. После этого восстановим в оставшихся цепочках вычеркнутые распознаватели.

В результате всех проведенных преобразований получаем каноническую схему Янова  $G_\beta^K(A, P)$ , эквивалентную исходной схеме  $G_\beta(A, P)$  и соответствующую управляемому графу программы  $\Gamma(B, D)$ .

По определению, на канонической схеме Янова в явном виде представлены корректные и некорректные фрагменты схемы программы, что позволяет при приведении ее к продуктивной программной структуре выявлять структурные дефекты программного кода, такие, как нарушения логики вычислений, неиспользуемые участки кода и зацикливаемые структуры.

В процессе приведения канонической схемы Янова  $G_\beta^K(A, P)$  к соответствующей ей продуктивной структуре выявляются и помечаются как структурные дефекты программы:

- все распознаватели с ложными условиями;
- все фрагменты схемы, соответствующие маршрутам, ведущим только на заглушки, и сами заглушки;

- все распознаватели, зацикливающиеся по минус-стрелкам на самих себя.

Перечень выявленных дефектов фиксируется и передается эксперту для проведения при необходимости дальнейшего анализа и включения в отчет.

Для получения продуктивной программной структуры все выявленные дефекты удаляются, в результате чего происходит устранение избыточности испытываемого программного обеспечения. Полученное представление программы является максимально компактным и наиболее подходящим для проведения дальнейших испытаний.

### **3.4 Формирование системы критериев для классификации выявленных дефектов как подобных НДВ**

В зависимости от характеристик и особенностей метрик сложности им ставятся в соответствие различные измерительные шкалы.

Номинальной шкале соответствуют метрики, классифицирующие программы на типы по признаку наличия или отсутствия некоторой характеристики без учета градаций. Порядковой шкале соответствуют метрики, позволяющие ранжировать некоторое характеристики путем сравнения с опорными значениями, т.е. измерение по этой шкале фактически определяет взаимное положение конкретных программ. Интервальной шкале соответствуют метрики, которые показывают не только относительное положение программ, но и то, как далеко они отстоят друг от друга. Относительной шкале соответствуют метрики, позволяющие не только расположить программы определенным образом и оценить их положение относительно друг друга, но и определить, как далеко оценки отстоят от границы, начиная с которой характеристика

может быть измерена. Для целей данного исследования наиболее подходят интервальная и относительная шкалы.

Оценка получаемых значений метрик в соответствии с целью исследования проводилась по двум вариантам.

В первом случае проводится вычисление значений метрик сложности в целом по исследуемой программе, что подразумевает расчет метрик по полному дизассемблированному коду исполняемого файла программы. Основной недостаток такого оценивания заключается в том, что получение корректных и достоверных данных при оценке значений метрик возможно только для относительно небольших программных проектов, выполненным одним программистом.

Применяя изложенный ранее подход, построенный на основе метрического базиса, представленного в таблицах 3.1 и 3.2, можно получить оценку квалификации программиста.

Оценка квалификации программиста выполняется на основе метрик размера программы  $m_{10}$  путем оценки отклонения рассчитанных мер реальной и теоретической длин программы. На основе полученных статистических данных можно ввести следующие интервальные зависимости между уровнем квалификации программиста и отклонением длины программы от метрического эталона (рассчитанной теоретической длины) (рис. 3.5):

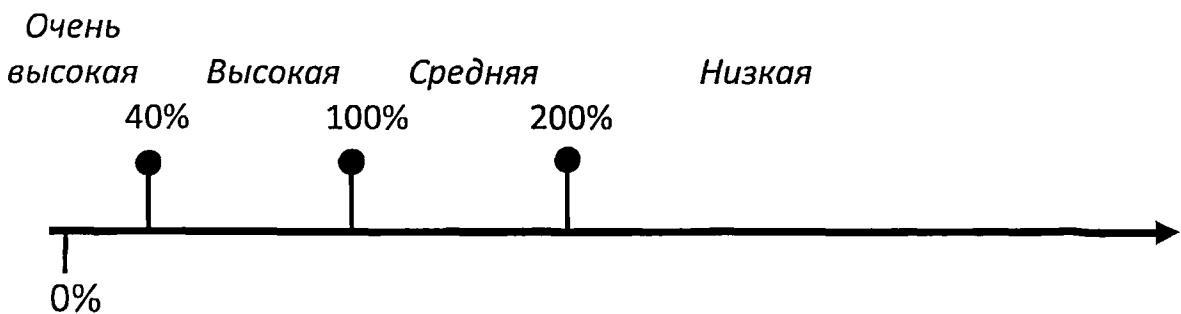


Рисунок 3.5 – Шкала оценки квалификации программиста.

Для оценивания характера относительных изменений между отдельными функциональными объектами (ФО) и ПС в целом для метрик информационной сложности был проведен экспериментальные исследования по расчету метрических показателей 3 модификаций программ при внесении в них программных закладок.

Внесение программных закладок может производиться в ручном или автоматизированном режиме. При работе в ручном режиме возможен подбор используемых в ней конструкций и информационных объектов таким образом, чтобы максимально «замаскировать» закладку путем имитации методов построения программы, примененных разработчиком, и внесением минимально возможных изменений в набор маршрутов выполнения программы. Автоматический режим создания закладки подразумевает использование уже ранее объявленных в программе операторов и operandов и некоторой имитацией стиля разработчика программы. Характерно, что при автоматизированной модификации программного кода наиболее сильно уменьшается значение метрики уровня реализации программы (более чем в 10 раз – рис. 3.6), что так же свойственно внедрению типовых закладок по вирусной технологии.

Представленные на рисунке 3.6 данные обобщают результаты исследования 16 типовых и уникальных программ. В процессе проведения эксперимента производилось вычисление значений набора метрик сложности для 3 вариантов каждой программы:

- 1) ее первоначального (исходного) состояния;
- 2) после внедрения в программу типовой программной закладки в автоматизированном режиме, реализующей останов (« зависание ») работы программы при определенных входных данных;
- 3) после ручного внедрения программной закладки, реализующей возможность удаленного получения несанкционированного доступа к системе.

Словарь программы



Рисунок 3.6 – Изменения значений метрик информационной сложности при внесении программных закладок

На рисунке совокупность значений метрик, помеченная цифрой 1, соответствует усредненным значениям исследуемых программ в их первоначальном состоянии: словарь программы равен в среднем 22 операндам и операторам, уровень реализации программы соответствует ~0,02, интеллектуальное содержание ~16.

Результаты измерения значений метрик после внедрения типовой программной закладки в автоматизированном режиме помечены цифрой 2. Учитывая особенности внедрения типовой программной закладки, необходимо отметить, что при неизменном словаре программы происходит значительное увеличение среднего размера дизассемблированного (соответственно, и исполняемого) кода программ (в 7 раз). Как следствие

появления подобной значительной избыточности кода, значение уровня реализации программы многократно уменьшается (в 50 раз), а значение метрики интеллектуального содержания ощутимо увеличивается (в среднем на 15%).

При внесении в программу созданной «вручную» уникальной программной закладки, т.е. максимально «подстроенной» под существующую реализацию программы, были рассчитаны значения метрик, представленные на рисунке графиком, помеченным цифрой 3. В силу того, что закладка такого рода максимально замаскирована, метрика уровня реализации практически всегда соответствует исходной программе. Словарь программы при этом незначительно расширяется, что объясняется применением небольшого количества дополнительных операторов, не используемых в исходной программе. Соответственно, при добавлении в программу дополнительных фрагментов происходит и увеличение объема исполняемого кода в среднем в 1,8 раза. Среднее значение метрики интеллектуального содержания увеличилось в данном случае на 56%.

Таким образом, в проведенной серии экспериментов четко прослеживается зависимость интеллектуального содержания программы от наличия в ней программной закладки. Это объясняется двумя причинами: во-первых, наличием у внедряемого фрагмента кода собственного интеллектуального содержания, во-вторых, прямой зависимостью между размером объема программы и значения ее интеллектуального содержания. Следовательно, при наиболее чувствительным показателем наличия в программе чужеродного кода (программной закладки) является метрика, характеризующая интеллектуальное (информационное) содержание.

Полученные в ходе проводимого эксперимента результаты показали, что наблюдаемые относительные изменения значений метрик инвариантны к размеру исследуемой программы или отдельного участка кода.

Следовательно, сравнение и анализ рассчитанных для отдельных участков кода и/или функциональных объектов значений выбранной взаимоувязанной тройки метрик, особенно метрики интеллектуального содержания, позволяют более точно локализовать и, в итоге, выявлять недекларированные возможности программных средств, классифицируемые как программные закладки.

Для формирования критерия, основанного на сложности управляющего графа программы были проведены экспериментальные исследования 48 типовых и уникальных программ и для них рассчитаны значения метрики Джилба и метрики точек пересечения.

Такая процедура может выполняться на первом этапе проведения статического анализа – после проведения лексико-семантического анализа, заключающегося в выделении в коде условно законченных функций (процедур) поиске и параллельном выявлении в них подозрительных и опасных сигнатур либо сразу после дизассемблирования исполняемого файла. При этом необходимо учитывать, что первоначальное состояние программного средства и составляющих его ФО является, как правило, достаточно обфускированным. В связи с этим рассчитанные значения метрик сложности не всегда информативны и достоверно отражают удовлетворение критериям. Тем не менее, в качестве экспресс-исследования получение результатов расчета метрик на данном этапе и проведение их сравнительного анализа позволяет выявлять наиболее потенциально опасные в качестве НДВ функциональные объекты.

Для получения более полной и достоверной информации необходимо выполнить расчет значения метрики Джилба и метрики точек пересечения после «очистки» программы путем приведения управляющего графа к продуктивной структуре на базе канонического вида схемы Янова, т.е. устранения части дефектов программы, выражавшихся, в первую очередь, зацикливанием, избыточностью и нарушением свойств принятой

программной структуры. Вычисление значений метрик сложности по каждому функциональному объекту после проведения таких преобразований дает возможность наиболее достоверно оценить информационные и топологические характеристики программы. Как следствие, становится возможным более точное оценивание возможности наличия / отсутствия и локализация потенциальных НДВ.

Кроме абсолютных значений метрик было рассчитано также отношение среднеквадратического отклонения к математическому ожиданию метрик Джилба и точек пересечения на этих двух этапах исследования программы. Таким образом, условие классификации выявленных дефектов как подобных НДВ и потенциально опасных программных конструкций определяется по формуле:

$$\frac{3\sigma_i}{m_i} \geq k_{zp_i}, \text{ где } i = 1, 5 \quad (3.23)$$

На основании анализа полученных экспериментальных данных было определено, что граничные значения  $k_{zp_i}$  для метрик Джилба и точек пересечения зависят от сечений, в которых они рассчитываются. Для «очищенных» или свободных от дефектов программ экспериментально были получены следующие «жесткие» граничные значения:  $k_{zp1} = 0,07$ ;  $k_{zp5} = 0,005$ . С ростом количества исследованных программ и объема статистических данных эти граничные значения могут уточняться.

На основе полученных экспериментальных данных были сформулированы критерии для классификации выявленных дефектов как подобных НДВ и потенциально опасных программных конструкций. Это следующая совокупность критериев:

- 1) характер относительных изменений между отдельными функциональными объектами (ФО) и ПС в целом метрик информационной сложности: теоретической и реальной длины программы, метрики

интеллектуального содержания и ее составляющих, в частности, уровня реализации программы;

2) превышение для метрик Джилба и точек пересечения усредненного по ФО значения отношения среднеквадратического отклонения к математическому ожиданию, рассчитанных для всех ФО, некоторых граничных значений, установленных экспериментально.

Дополнительным критерием может служить следующее отношение:

$$\frac{(\sigma_i / \bar{m}_i)_{\text{продуктивн}}}{(\sigma_i / \bar{m}_i)_{\text{дизассембл}}} \geq 1 \quad (3.24)$$

На практике это соотношение больше единицы и показывает степень «очищения» программы от простейших дефектов.

### Выводы по главе 3

Глава посвящена разработке метода выявления дефектов, подобных НДВ, с использованием структурированных метрик сложности. Сущность метода состоит в формировании метрического базиса как совокупности структурированных метрик топологической и информационной сложности, соотнесенных с конструкциями языка ассемблер; последующем их ранжировании путем экспертного оценивания; приведении логической модели программы к продуктивной структуре, построенной на основе канонического представления схем Янова; экспериментальном определении «чувствительности» и корреляции выбранных компонент метрического базиса к изменениям программного кода, и, на этой базе, разработке системы критериев оценивания безопасности функциональных объектов (участков программного кода).

Для универсализации разработанного метода в качестве исходных данных рассматривается дизассемблированный код программы, т.к.

оценивание программ на уровне дизассемблированного кода является наиболее универсальным, поскольку может проводиться как при наличии исходных текстов программ, так и в их отсутствие.

В качестве метрического базиса выбрана совокупность метрик топологической и информационной сложности, исходя из их смыслового содержания. На основе экспертного оценивания с использованием правил комбинирования теории свидетельств были получены обобщенные весовые коэффициенты метрик сложности, составляющих выбранный метрический базис, и проведено их ранжирование. В результате наиболее предпочтительными являются метрические характеристики длины текста, метрики Джилба и точек пересечения, а также метрика интеллектуального содержания.

При проведении эксперимента были исследованы 48 типовых и уникальных программ и рассчитаны их метрические характеристики, при этом помимо абсолютных значений метрик оценивались относительные отклонения по участкам кода, математическое ожидание и среднеквадратичное отклонение в рамках исследуемой программы. Полученные данные использованы для формирования критериев распознавания признаков дефектов программ, подобных НДВ, и подготовки рекомендаций по их устранению.

Проведенные экспериментальные исследования выбранных метрик сложности показали, что для выявления программных закладок требуется предварительно устраниć большую часть дефектов, не относящихся к НДВ, путем преобразования управляющего графа программы в продуктивную структуру, построенную на основе канонического представления схем Янова. Представление программы в виде логической схемы Янова позволяет явным образом выявлять часть дефектов программы, выражаяющихся, в первую очередь, зацикливанием, избыточностью и нарушением свойств принятой программной структуры.

Процесс приведения программы к продуктивной структуре на основе канонического представления схем Янова подробно описан в данной главе.

На основе полученных экспериментальных данных были сформулированы критерии для классификации выявленных дефектов как подобных НДВ и потенциально опасных программных конструкций. Это следующая совокупность критериев:

- 1) характер относительных изменений между отдельными функциональными объектами и программным средством в целом метрик информационной сложности: теоретической и реальной длины программы, метрики интеллектуального содержания и ее составляющих, в частности, уровня реализации программы;
- 2) превышение для метрик Джилба и точек пересечения усредненного по функциональным объектам значения отношения среднеквадратического отклонения к математическому ожиданию, рассчитанных для всех функциональных объектов, некоторых граничных значений, установленных экспериментально.

## **4 МЕТОДИКА ВЫЯВЛЕНИЯ НЕДЕКЛАРИРОВАННЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММ И ПРАКТИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ СТРУКТУРИРОВАННЫХ МЕТРИК СЛОЖНОСТИ**

### **4.1 Место и роль метрических оценок в системе статического анализа программных средств**

Одной из основных процедур статического анализа исходных текстов программ является предварительный контроль качества разработки программы на основе количественного оценивания выбранных метрик сложности. При исследовании программного средства с использованием метрик сложности в качестве анализируемых кодов может выступать дизассемблированный код испытываемых исполняемых файлов. Это является одним из преимуществ рассматриваемого подхода, т.к. при его использовании нет необходимости иметь в наличии исходные тексты программ на языках высокого уровня, что связано с максимальной конфиденциальностью информации. Кроме того, в этом случае решается проблема, связанная с возможными несовершенствами компилятора, т.к. применение средств компиляции оказывает существенное воздействие на качество программы.

Дополнительно, для повышения эффективности статического анализа программ в рамках сертификации по контролю отсутствия недекларированных возможностей по 1 – 3 уровню контроля и построения автоматизированных инструментальных средств для задач сертификации программного обеспечения по требованиям безопасности информации, необходимо объединить качественные и количественные показатели статического анализа путем расчета метрик оценки программного средства, получения метрической оценки управляющего графа и каждой функции (уровень языка, уровень программы, цикломатическое число и др.).

Методы и инструментальные программные средства, обеспечивающие проведение статического анализа, предусматривают получение таких графических и метрических характеристик, как модульная структура программного средства, логическая структура отдельного программного модуля, характеристики текста программы.

Модульная структура представляется в виде графа вызовов, списка путей вызовов, матрицы вызовов и достижимости, точек вызовов и метрик иерархии вызовов.

Логическая структура представляется в виде графа управления, путей тестирования и метрик структуры управления.

Характеристики текста программ включают в себя статистические данные о комментированности программы, текстовые метрики Холстеда и другие метрики информационной сложности.

Проведённые исследования, представленные в работах [20 – 25, 40, 63] и других работах автора, показали недостаточность приведённых характеристик, и, как следствие, необходимость использования метрик сложности для оценки соответствия программ требованиям качества и безопасности, в том числе выявления НДВ. В частности, предлагается использование структурированных метрик топологической и информационной сложности в системе статического анализа (рис. 4.1).

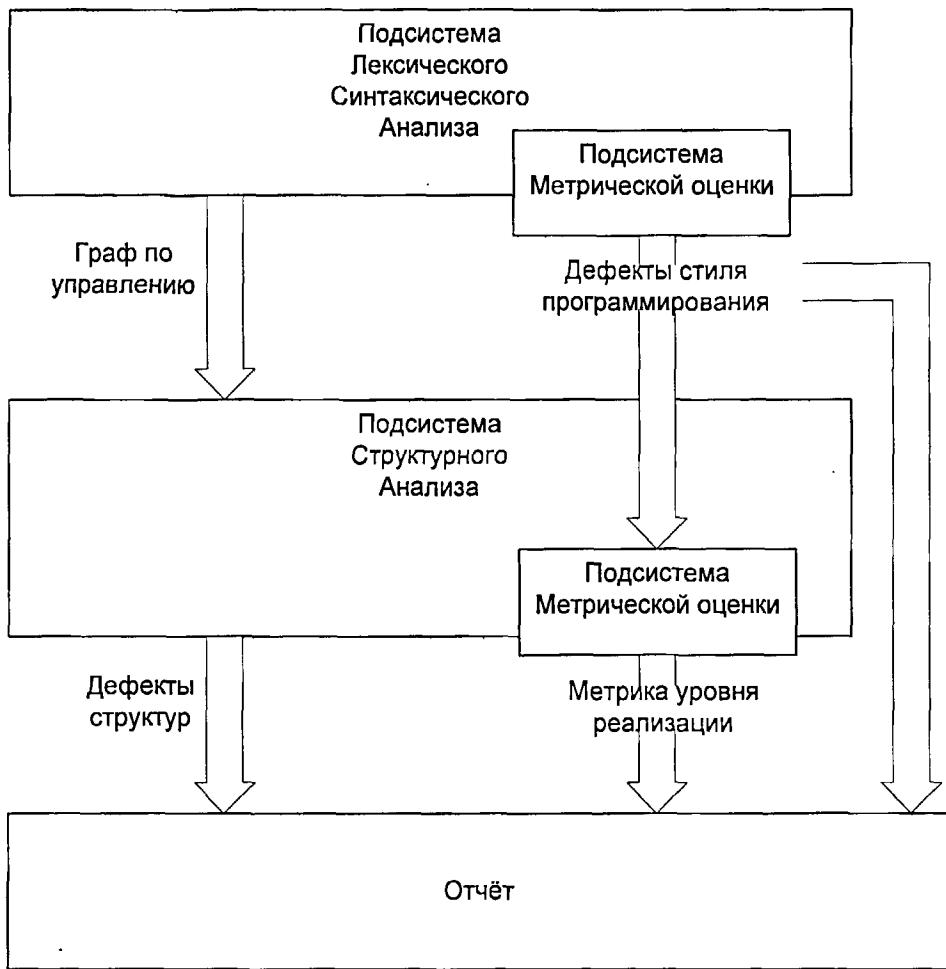


Рисунок 4.1 – Взаимодействие подсистем статического анализа исходных текстов программ с использованием метрических оценок

Представленная на рисунке подсистема лексического синтаксического анализа осуществляет разбор исходного текста программ с целью формирования графового представления управляющих структур и количественной оценки стиля программирования в каждой вершине графа, что позволит впоследствии выявить его искажения. Подсистема структурного анализа на основе графа по управлению и полученных оценок позволяет выявить дефекты управляющей структуры, а также оценить общий и соответствующий вершинам графа уровень реализации программы, позволяющий сформировать первичное суждение о качестве исполнения программ.

Предлагаемая подсистема метрической оценки реализует измерение на основе выбранных метрик и оценивание результатов разработанных правил, что позволяет выполнить:

- ранжирование функций и процедур по отклонению от доверенных интервалов;
- разбиение процедур и функций на содержащие дефекты и классифицируемые как НДВ.

Существуют и другие возможности применения метрических оценок в рамках статического анализа программного обеспечения.

На рисунке 4.2 представлены существующие в настоящее время методы статического контроля [63].

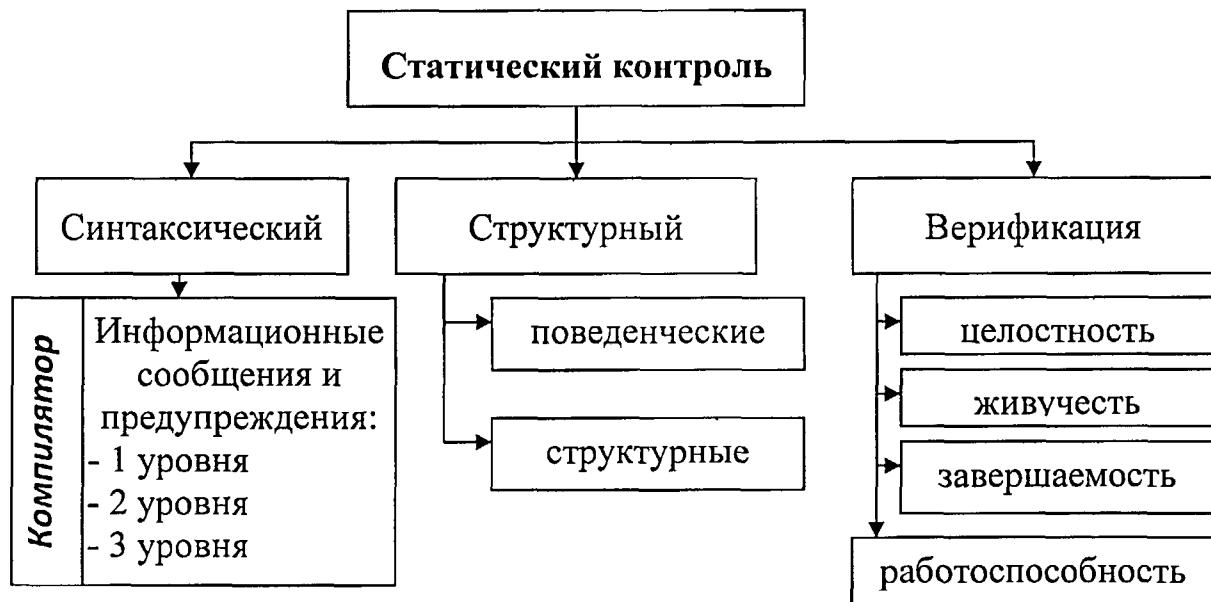


Рисунок 4.2 - Статические методы контроля программ на отсутствие НДВ

Проведение представленных методов статического контроля (структурный анализ, синтаксический анализ, верификация) возможно только при наличии семантической составляющей, т.е. при наличии исходных текстов программ. Применение метрик сложности позволяет реализовать данные методы при отсутствии семантического означивания используемых функций и процедур. Таким образом, с использованием

дизассемблированного кода становится возможным применение методов статического анализа для гораздо более широкого спектра программных средств на различных этапах их жизненного цикла.

Реализация метрического контроля обеспечивает автоматизированную верификацию статического кода по целостности, работоспособности и завершаемости программ; о живучести с позиции метрик возможно судить только косвенно. Применение метрик в верификации необходимо для сравнения программ, идентификации их модулей, а так же вынесение суждений о подозрительности на наличие недекларированных возможностей [20].

Один из основных принципов применения метрик сложности заключаются в выделении процедур, имеющих одинаковые значения метрик, в первую очередь, метрик целостности информационного содержания (метрики Холстеда, Хансена). Для выделенных процедур проводится сравнение значений остальных метрик. При существенных различиях в значениях метрик топологической сложности, характеризующих работоспособность статического кода, можно сделать вывод о структурной избыточности выделенной процедуры, т.е. о наличии НДВ. При мере сходства в значениях остальных метрик выделенных процедур, стремящейся к единице, также можно сделать вывод о наличии НДВ как результата реализации двух или более максимально подобных процедур.

С другой стороны, проводя оценивание программного обеспечения с помощью некоторых других метрик, характеризующих топологическую и информационную сложность, получение одинаковых или, по крайней мере, достаточно близких значений свидетельствует о достаточно высокой вероятности отсутствия НДВ в данном программном средстве.

Каждому языку программирования соответствует свой рассчитанный уровень языка. Значение метрики «Уровень языка», не соответствующее

необходимому значению для «родного» языка реализации программы, сигнализирует о наличии в программе вставок на другом языке программирования, например, ассемблере. Обнаружение вставки на языке ассемблер должно однозначно вызывать подозрение о наличии программных закладок.

Каждый программист имеет свои особенности написания программ, которые характеризуют индивидуальные особенности его как разработчика программы. В таком случае применимы метрики сложности, которые непосредственно связаны с различными количественными показателями, позволяющими оценивать стиль программирования. Для этого используются дополнительные метрические показатели, характеризующие словарную и смысловую насыщенность программы: объем программы ( $V$ ), метрики трудности понимания программ, интеллектуальное содержание программы ( $I$ ), метрики размерности комментариев в программе и др. Эти дополнительные метрики очень важны при анализе исходных текстов на отсутствие недекларированных возможностей (НДВ). Именно они являются показателями, с помощью которых можно отследить изменение стиля программирования, т. е. провести декомпозицию исходных текстов с целью выявления НДВ. Помимо самого стиля программирования, с помощью метрик сложности можно вычислить и уровень разработки конкретного программного средства. Для этого применимы такие метрики сложности программы, как показатель уровня программы ( $L$ ), метрика уровня языка выражения ( $\lambda^*$ ) и другие.

Исследования показали, что при повышении уровня структурированности программы значения метрик увеличиваются, но динамика изменения метрических показателей разная. Выполняя вычисление приведенных метрик в каждой вершине управляющего графа, можно отслеживать изменение стиля программирования, уровня

реализации программ и других показателей, изменение которых в программе, написанной одним программистом, свидетельствует о возможном наличии в ней НДВ.

Таким образом, используя системный подход, эксперт применяет на основе разработанных правил предложенную в главе 3 метрическую модель, построенную на основе структурированного метрического базиса. При получении значений метрик базиса для совокупности модулей и функциональных объектов, входящих в состав исследуемого программного средства, не удовлетворяющих требованиям к отсутствию НДВ, делается вывод о возможном наличии НДВ в данном программном обеспечении.

#### **4.2 Методика выявления недекларированных возможностей программ с использованием структурированных метрик топологической и информационной сложности**

Методика контроля исполняемого кода предназначена для выявления дефектов и НДВ программ. Методика включает этап предварительного анализа исследуемого программного обеспечения, построение управляющего графа программы и приведение его к каноническому виду на основе схем Янова, расчете совокупности метрик для всего исполняемого кода в целом и для отдельных функциональных объектов, анализ полученных результатов расчета и, на их основе, формирование заключения о наличии или отсутствии НДВ и дефектов программного средства [22]. . . .

Обнаруженные в процессе анализа дефекты и НДВ используются для уточнения классификации признаков дефектов и принятия предупредительных мер, необходимых при обеспечении информационной безопасности испытываемого программного средства, а также как

доказательства наличия или отсутствии НДВ при проведении сертификационных испытаний.

Методика выявления недекларированных возможностей программ с использованием структурированных метрик топологической и информационной сложности, разработанная на основе предложенного в главе 3 метода, представлена на рисунке 4.3.

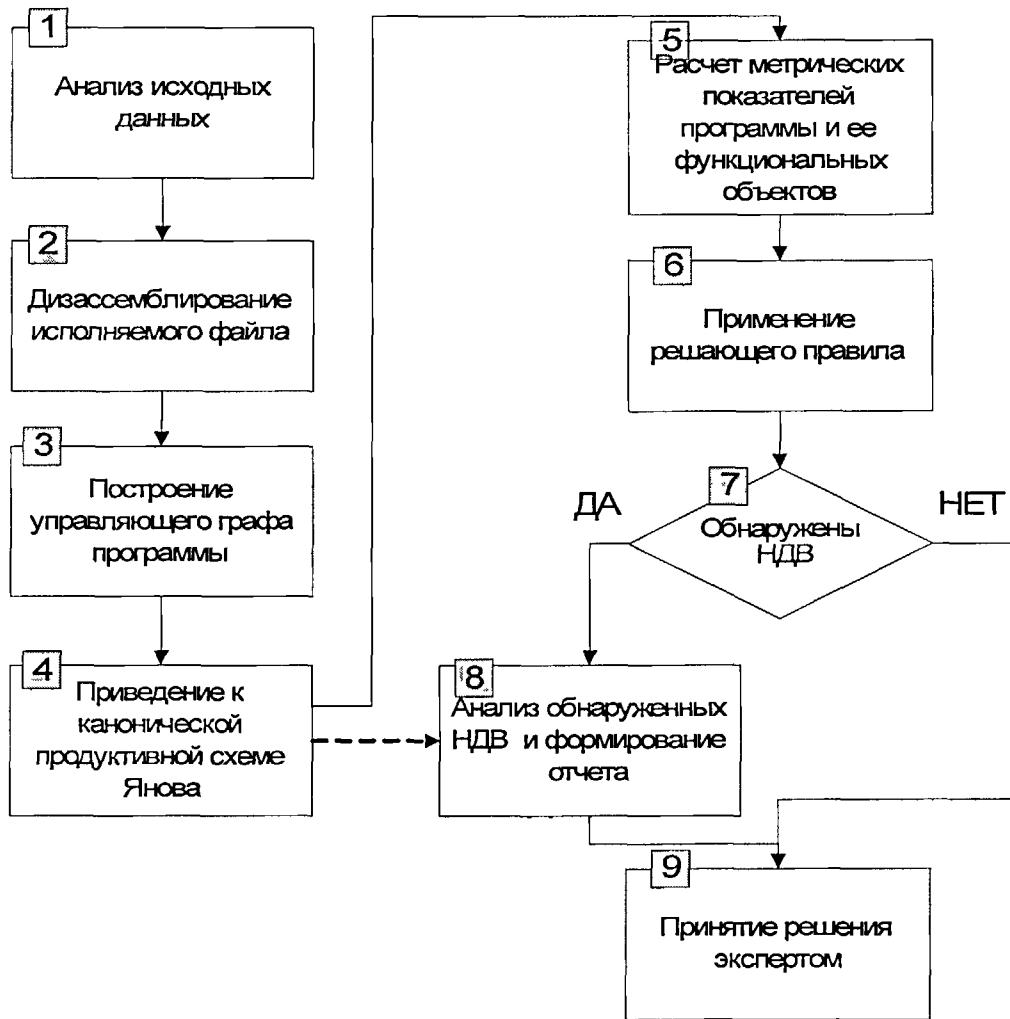


Рисунок 4.3 – Методика выявления НДВ с использованием метрик сложности

ШАГ 1. Проводится предварительный анализ испытываемого программного средства, включая и программную документацию, для ознакомления с функциональностью и спецификой данного программного средства для выявления критичных с точки зрения информационной

безопасности функций и функциональных объектов. Также на данном этапе проводится лексический синтаксический анализ исходных кодов программ (при их наличии) для выявления ошибок программирования.

ШАГ 2. Включает в себя блоки 2 и 3 на рис. 4.3. Дизассемблирование исследуемого исполняемого файла проводится с помощью стандартных инструментальных средств, например, пакета IDA Pro, для дальнейшего преобразования полученного кода в управляющий граф программы согласно принятым принципам построения графа по управлению.

ШАГ 3. Управляющий граф программы, полученный на предыдущем шаге, путем проведения эквивалентных преобразований приводится к виду логической схемы Янова на основе нахождения минимального покрытия всех путей управляющего графа и определения предикатных условий соответствующей ему схемы с помощью символического выполнения программы по найденным маршрутам. Далее, используя систему аксиом и правил вывода эквивалентного преобразования алгебры Янова, полученная логическая схема преобразуется в каноническую продуктивную структуру. На данном этапе происходит выявление потенциально зацикливаемых и тупиковых участков кода, а также участков кода, содержащих нарушения логики вычислений. Полученные данные о обнаруженных дефектах испытываемого программного обеспечения при необходимости направляются для дополнительного анализа эксперту.

ШАГ 4. Расчет совокупности метрических показателей топологической и информационной сложности, в общем случае определенных на этапе формирования метрического базиса (п. 3.2), производится для всего исполняемого кода в целом и для выделенных на этапе 1 отдельных функциональных объектов на основе полученной на предыдущем шаге канонической продуктивной схемы. В условиях проведения сертификационных испытаний программного комплекса для

повышения оперативности проведения испытаний рекомендуется сокращение количества рассчитываемых метрик сложности с учетом проведенного в п. 3.2 ранжирования.

ПОКАЗАТЕЛЬ	1	2	....	8	МАТЕМАТИЧЕСКОЕ ОЖИДАНИЕ	СКО
МЕТРИКА ДЖИЛБА	2,84277	2,825		2,828	2,831	0,041
МЕТРИКА ТОЧЕК ПЕРЕСЕЧЕНИЯ	5586	5585		5583	5585	5

Рисунок 4.4 – Примеры рассчитанных показателей в метрическом базисе структуры программ

Помимо абсолютных значений метрик рассчитывается вектор математического ожидания и среднеквадратическое отклонение в рамках исследуемой программы. Полученные данные используются на следующем этапе для распознавания признаков НДВ, дефектов программ и подготовки рекомендаций по их устранению.

ШАГ 5. На основании анализа полученных результатов осуществляется выявление метрических нарушений для отдельных участков исследуемого кода в соответствии с сформулированными в п. 3.4 критериями.

ШАГ 6. При наличии участков кода, для которых выявлено наличие метрических нарушений, попадающих в категорию «подобные НДВ», формируется отчет с указанием таких объектов, как требующих дополнительного исследования экспертом. В данный отчет также попадают выявленные на этапе 3 дефекты программного обеспечения в случае, если они локализованы в определенных ранее критичных с точки зрения информационной безопасности функциональных объектах. Дополнительные исследования могут проводиться следующими способами:

- путем расчета полного метрического базиса для подозрительных участков кода и анализа полученных абсолютных и относительных значений;
- с помощью различных инструментальных средств проведения статического анализа только для рассматриваемых фрагментов, что существенно уменьшает объем и повышает достоверность получаемых в этом случае результатов;
- в случае нечеткости и неоднозначности выявления признаков НДВ – эвристически экспертом на основе определенных ранее критичных маршрутах управляющего графа и базы практических знаний.

На основании проведенного анализа выявленных дефектов формируется итоговый отчет с указанием местоположения обнаруженных возможных НДВ и определенных признаков их классификации.

**ШАГ 7.** По результатам анализа полученного отчета экспертом по каждому случаю указанной возможной НДВ делается вывод о вредоносном или случайном характере дефекта, наличии / отсутствии НДВ или потенциально опасных конструкций, подозрительных на НДВ.

Для оценивания эффективности предлагаемой методики при проведении испытаний при контроле отсутствия НДВ программных средств за счет использования предлагаемой совокупности метрик сложности был использован прототип программной реализации подсистемы метрического анализа. С его помощью проведены испытания 14 программных проектов разного объема с целью оценки оперативности и полноты выявления НДВ при использовании разработанного метода в автоматизированном режиме. В таблице 4.1 представлены усредненные результаты по затраченному экспертом времени для проведения статического анализа испытываемых программных средств, анализа и обработки результатов.

Таблица 4.1. Среднее время проведения испытаний экспертом.

Среднее время проведения испытаний	Объем исполняемого ПС		
	До 100 Кб	100 Кб – 10 Мб	10 Мб – 100 Мб
Стандартными инструментальными средствами	120 час	280 час	480 час
Инструментальными средствами с применением метрического анализа	80 час	160 час	200 час

Согласно выполненным расчетам, затраты времени на проведение статического анализа программного средства экспертом в среднем сократились в 1,5 – 2 раза за счет более четкой локализации участков кода, являющихся подозрительными на НДВ. Также при этом достигнута большая полнота и достоверность выявления НДВ за счет вычисления значений метрик сложности для соответствующих участков кода и оценивания отклонений полученных значений метрик согласно предложенной в главе 3 системе критериев.

## Выводы по главе 4

В главе определены место и роль метрических оценок в системе статического анализа программных средств. Для повышения эффективности статического анализа программ в рамках сертификации по контролю отсутствия НДВ по 1 – 3 уровню контроля и построения автоматизированных инструментальных средств для задач сертификации ПС по требованиям безопасности информации, предлагается объединить качественные и количественные показатели статического анализа путем расчета метрик оценки программного средства, получения метрической оценки управляющего графа и каждой функции. Исходя из построенного ранее метрического базиса, связанного с логическими моделями структуры и свойств вычислимости программы, предлагается ввести подсистему

метрической оценки на этапах проведения лексического синтаксического анализа и структурного анализа испытываемого программного средства. Предлагаемая подсистема метрической оценки реализует измерение на основе выбранных метрик и оценивание результатов разработанных правил, что позволяет выполнить ранжирование функций и процедур по отклонению от доверенных интервалов и разбиение процедур и функций на содержащие дефекты, классифицируемые как некритичные и подобные НДВ.

Методика выявления НДВ программ в дизассемблированном коде с использованием структурированных метрик топологической и информационной сложности и включением подсистем метрического анализа на этапах лексико-синтаксического и структурно-семантического статического анализа разработана на основе описанного в главе 3 метода и учитывает предложения по реализации подсистемы метрических оценок при проведении статического анализа программ. Дополнительно в методику введены практические рекомендации по использованию метрик сложности в процедурах подтверждения соответствия программных средств по требованиям качества и информационной безопасности.

Согласно оценке временной эффективности испытаний при контроле отсутствия НДВ программных средств за счет использования предлагаемой совокупности метрик сложности, затраты времени на проведение статического анализа программного средства экспертом сократились в 1,5 – 2 раза за счет более четкой локализации участков кода, являющихся подозрительными на НДВ. Также при этом достигнута большие полнота и достоверность выявления НДВ за счет вычисления значений метрик сложности для соответствующих участков кода и оценивания отклонений значений метрик на основе предложенной системы критериев.

## ЗАКЛЮЧЕНИЕ

В результате проведенного исследования были разработаны метод и методика выявления в программных средствах дефектов, подобных НДВ, с использованием подсистемы метрического анализа. Применение разработанного метода позволяет повысить полноту и сократить временные затраты на проведение испытаний при контроле наличия/отсутствия НДВ программных средств.

При решении частных задач были получены следующие результаты:

1. Выполнен анализ объекта исследования и современных методов исследования программ, верификации и выявления НДВ программных средств. В частности, проанализированы нормативная база, критериальный аппарат и методы подтверждения соответствия программных средств по требованиям безопасности информации, и выявлены их основные недостатки: значительная вычислительная сложность процедур статического и динамического анализа, недостаточность декларированных способов проведения некоторых предписанных проверок, сложность проведения испытаний в отсутствие исходных текстов программ.

2. Предложен и обоснован подход к выявлению дефектов программ, подобных НДВ, базирующийся на анализе формальных семантик, вычислительных структур и свойств программ, структуризации моделей представления программы и оценивающих их метрик сложности.

3. Разработан метод выявления дефектов, подобных НДВ, в программных средствах, основанный на формировании метрического базиса в виде структурированных метрик топологической и информационной сложности, приведении управляющего графа программы к каноническому представлению схемы Янова и продуктивной структуре, построении системы критериев метрического оценивания безопасности функциональных объектов (участков программного кода) на базе результатов экспериментальных исследований.

4. Разработана методика выявления НДВ программ в дизассемблированном коде с использованием структурированных метрик топологической и информационной сложности и включением подсистем метрического анализа на этапах лексико-синтаксического и структурно-семантического статического анализа.

5. Описаны принципы построения и функционирования программной реализации подсистемы метрического анализа. Даны практические рекомендации по ее применению в системе формальной верификации и выявления НДВ программных средств, в том числе и для использования в работе испытательной лаборатории.

6. Проведено исследование оперативности и полноты выявления НДВ на этапе статического анализа с использованием подсистемы метрического анализа. Показано, что оперативность и полнота выявления НДВ повышается в 1,5 – 2 раза.

Таким образом, в ходе проведенного исследования были решены поставленные частные задачи и главная научная задача исследования и достигнута цель исследования, состоящая в повышении полноты и сокращении временных затрат на проведение испытаний при контроле наличия/отсутствия НДВ программных средств

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Федеральный закон «О техническом регулировании» № 184-ФЗ от 27.12.2002 г. [эл. ресурс <http://www.gost.ru/wps/portal/>]
2. Технический регламент о безопасности железнодорожного подвижного состава, утв. постановлением Правительства РФ № 524 от 15 июля 2010 г. [эл. ресурс <http://www.gost.ru/wps/portal/>]
3. Технический регламент о безопасности инфраструктуры железнодорожного транспорта, утв. постановлением Правительства РФ № 525 от 15 июля 2010 г. [эл. ресурс <http://www.gost.ru/wps/portal/>]
4. Технический регламент о безопасности высокоскоростного железнодорожного транспорта, утв. постановлением Правительства РФ № 533 от 15 июля 2010 г. [эл. ресурс <http://www.gost.ru/wps/portal/>]
5. Алферова З.В. Теория алгоритмов. – М.: Статистика. 1973. – 164 с.
6. Андерсон Р. Доказательство правильности программ. – М.: Мир, 1982. – 165 с.
7. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ. – М.: Мир, 1978. – 612 с.
8. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 2: Компиляция. – М.: Мир, 1978. – 487 с.
9. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. – М.: Издательский дом «Вильямс», 2001. – 768 с.
10. Барендргарт Х. Ламбда-исчисление. Его синтаксис и семантика: пер. с англ. – М.: Мир, 1985. – 606 с.
11. Богданов Д.В., Фильчаков В.В. Стандартизация жизненного цикла и качества программных средств: Учебное пособие. – СПб.: ГУАП, 2000. – 210 с.

12. Боэм Б., Браун Дж., Каспар Х., Липов М., Мак-Леод Г., Мерит М. Характеристики качества программного обеспечения. / пер. с англ. – М.: Мир, 1981. – 208 с.
13. Боэм Б.У. Инженерное проектирование программного обеспечения: пер с англ. – М.: Радио и связь, 1985. – 512 с.
14. Бэкус Дж. Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ: пер. с англ. Мартынюка В.В. // В кн.: Лекции лауреатов премии Тьюринга за первые двадцать лет. 1966 – 1985 гг. / под ред. Р. Эшенхерста. – М.: Мир, 1993. – С. 84 – 158.
15. Бэкус Дж. Алгебра функциональных программ: мышление функционального уровня, линейные уравнения и обобщенные определения // Математическая логика в программировании: Сб. статей 1980 – 1988 гг.: пер. с англ. – М.: Мир, 1991. – С. 8 – 53.
16. Вирт Н. Алгоритмы + структуры данных = программы. – М., Мир, 1985. – 336 с.
17. Волкова И.А., Руденко Т.В. Формальные грамматики и языки. Элементы теории трансляции. – М.: Издательский отдел факультета ВМиК МГУ им. М.В. Ломоносова, 1999. – 62 с.
18. Дал У., Дейкстра Э., Хоар К. Структурное программирование: пер. с англ. – М. Мир, 1975. – 247с.
19. Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978. – 275 с.
20. Диасамидзе С.В. Использование метрик сложности для логической оценки качества разработки программных средств // Интеллектуальные системы на транспорте: материалы I международной научно-практической конференции «ИнтеллектТранс-2011». - СПб.: ПГУПС, 2011. - С. 130 – 134.

21. Диасамидзе С.В. Комплексные сертификационные испытания программных средств по требованиям качества и безопасности информации // Информационные технологии на железнодорожном транспорте: Доклады тринадцатой международной научно-практической конференции «Инфотранс-2008». - СПб.: ПГУПС, 2008. - С. 127 – 131.
22. Диасамидзе С.В. Метод и методика выявления недекларированных возможностей программ с использованием структурированных метрик сложности // Известия Петербургского университета путей сообщения. - СПб: ПГУПС, 2012. – Вып. 3 (32). - С. 29 – 36.
23. Диасамидзе С.В. Принцип сертификационных испытаний программных средств // Автоматика, связь, информатика. – 2009, № 7. – С. 29 – 30.
24. Диасамидзе С.В. Проблемы сертификационных испытаний программных средств связи // Автоматика, связь, информатика. – 2009, № 2. – С. 31 – 32.
25. Диасамидзе С.В., Новиков В.А., Платонов А.А. Использование метрик сложности для оценивания качества программ // Информационные технологии на железнодорожном транспорте: Доклады двенадцатой международной научно-практической конференции «Инфотранс-2007». - СПб.: ПГУПС, 2008. – С. 70 – 72.
26. Донаху Д. Взаимодополняющие определения семантики языка программирования. / В кн.: Семантика языков программирования. – М.: Мир, 1980. – С. 222 – 394.
27. Ершов А.П. Введение в теоретическое программирование (беседы о методе). – М.: Наука, 1977. – 288 с.
28. Ершов А.П. Современное состояние теории схем программ // В сб.: Проблемы кибернетики, вып. 27. – М.: Наука, 1973. – С. 87 – 110.

29. Ершов А.П. Об операторных схемах Янова // В сб.: Проблемы кибернетики, вып. 20. – М.: Физматгиз, 1968. – С. 181 – 200.
30. Ершов Ю.Л. Определимость и вычислимость. – Новосибирск: Научная книга, 1996. – 300 с.
31. Ершов Ю.Л. Теория нумераций. – М.: Наука, 1977. – 416 с.
32. Изосимов А.В., Рыжко А.Л. Метрическая оценка качества программ. – М.: Изд. МАИ, 1989. – 96 с.
33. Казиев В.М. Введение в анализ, синтез и моделирование систем. – М.: Интернет-Университет Информационных технологий «Интуит.ру»; БИНОМ. Лаборатория знаний, 2008. – 248 с.
34. Кауфман В.Ш. Языки программирования. Концепции и принципы. – М., Радио и связь, 1993. – 432 с.
35. Клини С.К. Введение в метаматематику. – М.: ИЛ, 1957. – 526 с.
36. Клини С.К. Математическая логика. – М.: Мир, 1973. – 480 с.
37. Ключевский Б. Программные закладки // Системы безопасности, связи и телекоммуникаций – 1998, №22. – С. 60 – 66.
38. Коваленко И. И., Швед А. В. Оценка качества программных продуктов с использованием теории Демпстера-Шейфера // Наукові праці. Серія: Комп'ютерні технології. Випуск 148, Том №160. – Миколаїв: Редакційно-видавничий центр ЧДУ імені Петра Могили, 2011. – С. 22 – 26.
39. Корниенко А.А., Бубнов В.П. Проблемы подтверждения соответствия и сертификации программных средств информационно-управляющих систем железнодорожного транспорта // Сборник докладов XIII МНПК «Инфотранс-2008». – СПб.: ПГУПС, 2008. – С. 7 – 14.
40. Корниенко А.А., Диасамидзе С.В. Использование метрик сложности для распознавания недекларированных возможностей и оценивания качества программного обеспечения // Телекоммуникационные, информационные и логистические технологии на транспорте России: Сборник докладов четвертой международной научно-

практической конференции «ТелекомТранс-2008». – Ростов н/Д.: РГУПС, 2005. - С. 74 – 78.

41. Корниенко А.А., Диасамидзе С.В., Schäbe Hendrik, Tsypor Alexander. Сертификация программного обеспечения согласно европейским и российским стандартам безопасности для железнодорожных систем – сравнение // Интеллектуальные системы на транспорте: материалы I международной научно-практической конференции «ИнтеллекТранс-2011». - СПб.: ПГУПС, 2011. - С. 42 – 56.

42. Корниенко А.А., Диасамидзе С.В. Подтверждение соответствия и сертификация программного обеспечения по требованиям безопасности информации: Учебное пособие. – СПб.: ПГУПС, 2009. – 55 с.

43. Корниенко А.А., Сафонов В.И. Обязательное подтверждение соответствия программных средств железнодорожного транспорта // Транспорт Российской Федерации. – 2009, № 3(22). – С. 36 – 39.

44. Криницкий Н.А. Равносильные преобразования алгоритмов и программирование. – М.: Советское радио, 1970. – 304 с.

45. Куламин В.В. Методы верификации программного обеспечения / Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению «Информационно-телекоммуникационные системы». – М.: Институт системного программирования РАН, 2008. – 117 с.

46. Лаврищева Е.М., Петрухин В.А. Методы и средства инженерии программного обеспечения: Учебник. – М.: МФТИ (ГУ), 2006. – 304 с.

47. Липаев В.В. Тестирование программ. – М.: Радио и связь, 1986. – 296 с.

48. Ляпунов А.А. О логических схемах программ. // В сб.: Проблемы кибернетики, вып. 1. – М.: Физматгиз, 1958. – С. 46 – 74.

49. Маевский Д.А., Яремчук С.А. Оценка количества дефектов программного обеспечения на основе метрик сложности //

Электротехнические и компьютерные системы. – Одесса: ОНПУ, 2012. – № 07(83). – С. 113 – 120.

50. Майерс Г. Надежность программного обеспечения. / пер. с англ.; под ред. В.Ш. Кауфмана. – М.: Мир, 1980. – 360 с.

51. Манна З. Теория неподвижной точки программ // Кибернетический сборник, № 15. – М.: Мир, 1978. – С. 38 – 100.

52. Молчанов А.Ю. Системное программное обеспечение: учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.

53. Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. – М.: Радио и связь, 1988. – 256 с.

54. Платонов А.А., Горемыкин Д.В., Еремеев М.А., Ломако А.Г., Новиков В.А., Гnidко К.О. Метод обнаружения дефектов в бинарных дампах памяти. // Компьютерные системы. Проблемы информационной безопасности. – № 3. – СПб.: 2009. – С. 25 – 32.

55. Подловченко Р.И. О проблеме эквивалентных преобразований программ // Программирование. – 1986, № 6. – С. 3 – 14.

56. Подловченко Р.И. От схем Янова к теории схем программ // В сб.: Математические вопросы кибернетики, вып. 7. – М., Физматлит, 1998. – с. 281 – 302.

57. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация. 4-е изд. – СПб.: Питер, 2002. – 688 с.

58. Себеста Р.У. Основные концепции языков программирования. – М.: Издательский дом «Вильямс», 2001. – 672 с.

59. Серебряков В.А., Галочкин М.П., Гончар Д.Р., Фуругян М.Г. Теория и реализация языков программирования. – М.: МЗ Пресс, 2006. – 352 с.

60. Синицын С.В., Налютин Н.Ю. Верификация программного обеспечения: учебное пособие. – М.: Интернет-Университет

Информационных технологий «Интуит.ру»; БИНОМ. Лаборатория знаний, 2008. – 368 с.

61. Скотт Д.С. Области в денотационной семантике. // Математическая логика в программировании: Сб. статей 1980 – 1988 гг. / пер. с англ. – М.: Мир, 1991. – С. 56 – 118.
62. Смит Д. Дж., Симпсон К. Дж. Л. Функциональная безопасность. Простое руководство по применению стандарта МЭК 61508 и связанных с ним стандартов. – М.: Издательский дом «Технологии», 2004. – 208 с.
63. Теоретические и экспериментальные исследования методов статистического анализа исходных текстов прикладного программного обеспечения средств вычислительной техники и программного обеспечения средств технической защиты информации. / Корниенко А.А., Диасамидзе С.В., Еремеев М.А., Новиков В.А. – Отчет НИР, ч. 1, 2. – М.: ФГУП «ЗаштаИнфоТранс», 2008 – 2010.
64. Турчин В. Ф. Алгоритмический язык рекурсивных функций (РЕФАЛ). — М.: ИПМ АН СССР, 1968.
65. Турчин В.Ф. Метаалгоритмический язык. // Кибернетика. – 1968, № 4. – С. 116–124.
66. Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛЕ. // В сб.: Автоматизированная система управления строительством. Труды ЦНИПИАСС, № 6. – М.: ЦНИПИАСС, 1974. – С. 36 – 68.
67. Филд А., Харрисон П. Функциональное программирование. – М.: Мир, 1993. – 637 с.
68. Холстед М.Х. Начала науки о программах / пер. с англ. В.М. Юфы. – М.: Финансы и статистика, 1981. – 128 с.
69. Яблонский С. В. Элементы математической кибернетики. – М.: Высшая школа, 2007. – 191 с.

70. Янов Ю.И. Предельно полная система правил эквивалентных преобразований для программ, вычисляющих всюду определенные функции. // В сб.: Проблемы кибернетики, вып. 37. – М.: Наука, 1980. – С. 215 – 238.
71. Янов Ю.И. О локальных преобразованиях схем алгоритмов. // В сб.: Проблемы кибернетики, вып. 20. – М.: Физматгиз, 1968. – С. 201 – 216.
72. Янов Ю.И. О логических схемах алгоритмов. // В сб.: Проблемы кибернетики, вып. 1. – М.: Физматгиз, 1958. – С. 29 – 53.
73. Church A. An Unsolvable Problem of Elementary Number Theory // Bulletin of the American Mathematical Society. 1935. – Vol. 41. – P. 332–333.
74. Cook W., Hill W.L., Canning P.S. Inheritance is not subtyping. // In: Proc. 17th ACM Symposium on Principles of Programming Languages, Jan. 1990. – P. 125 – 135.
75. Dijkstra T.W. Finding the Correctness proof of a concurrent program // Proc.Konf. Nederland Acad. Wetenach, 1978. – Vol. 81, № 2. – P. 207–215
76. Floyd R. Assigning meaning to programs // Mathematical Aspects Computer Science. Amer. Math. Soc. 1967. – Vol. XIX. – P. 19 – 32.
77. Hoare C.A. An axiomatic basis for computer programming // Communications ACM. 1969. – Vol.12, № 10. – P. 576 – 583.
78. Hoare C.A. Proof of correctness of data representation // Acta Informatica, № 1(4), 1972. – P. 214–224.
79. Knuth D.E. Semantics of Context-Free Languages // Mathematical Systems Theory, 1968. – Vol. 2, № 2. – P. 127 – 146.
80. McCabe T.J. A complexity measure // IEEE Transactions on Software Engineering, December, 1976. – Vol. SE-2, № 4. – P. 308 – 320.
81. McCarthy J., Abrahams P.W., Edwards J., Hart T.P., Levin M.I. LISP 1.5 Programmer's Manual. – M.I.T. Press, Cambridge, Massachusetts, 1965.

82. Sommerville I. Software Engineering . – Hardcover, Addison Wesley Longman, 1996. – 742 p.

83. Scott D.S. Lectures on a mathematical theory of computations. – Oxford University Computing Laboratory Technical Monograph. PRG-19, 1981. – 148 p.

84. Stoy J. E. Denotational semantics: the Scott-Strachey approach to programming language theory. – M.I.T. Press, Cambridge, Massachusetts, 1977.