Федеральное государственное бюджетное образовательное учреждение
высшего образования «Сибирский государственный университет науки
и технологий имени академика М.Ф. Решетнева»

На правах рукописи

Сарамуд Михаил Владимирович

МОДЕЛЬНО – АЛГОРИТМИЧЕСКОЕ ОБЕСПЕЧЕНИЕ АНАЛИЗА ОТКАЗОУСТОЙЧИВОСТИ ПРОГРАММНЫХ КОМПЛЕКСОВ ВСТРАИВАЕМЫХ СИСТЕМ УПРАВЛЕНИЯ РЕАЛЬНОГО ВРЕМЕНИ

05.13.01 – Системный анализ, управление и обработка информации
(космические и информационные технологии)

ДИССЕРТАЦИЯ

на соискание ученой степени кандидата технических наук

Научный руководитель
доктор технических наук, профессор
Ковалев И.В.

Красноярск – 2018

Оглавление

Введение

Актуальность работы. В последние годы активно развиваются отрасли, требующие надежных, отказоустойчивых систем управления реального времени. К их числу относятся высокотехнологичные производства, использующие композитные и опасные материалы, автономные беспилотные объекты - от мультироторных систем до автомобилей с функцией автопилота и моторизованных кресел с голосовым управлением для людей с ограниченными возможностями. Программное обеспечение (ПО) является неотъемлемой частью современных систем управления, однако, лишь простое ПО может быть гарантированно создано без ошибок. Современное ПО систем управления используется для решения все более сложных задач, лавинообразно возрастает объем обрабатываемой информации. С возрастанием сложности ПО растет и вероятность возникновения ошибок в нем.

Традиционно, методы повышения надежности встраиваемых систем управления (СУ) были основаны на повышении надежности аппаратных средств, для чего существуют классические инструменты и подходы, обеспечивающие требуемый уровень надежного функционирования СУ, включая применение дублирования компонентов СУ, что не имеет смысла для ПО, поскольку в этом случае будут дублироваться и ошибки в программах. В настоящее время актуальна разработка и применение методов программной отказоустойчивости, что связано с усложнением и укрупнением программных комплексов, увеличением количества функциональных модулей СУ, что приводит к росту вероятности отказа системы из-за отказа одного из модулей. Рост же надежности современных аппаратных средств СУ обеспечивается благодаря развитию технологических процессов производства радиоэлектронной аппаратуры, а благодаря снижению стоимости упрощается дублирование аппаратных компонентов.

Особенно критична надежность ПО в системах реального времени. Если в обычной системе в случае возникновения сбоя есть время, чтобы его обработать, восстановить состояние системы до сбоя и повторить вычисления, то в случае работы в режиме реального времени нет возможности восстановления и

повторного исполнения задачи, а по истечении времени на выполнение данная задача будет завершена принудительно.

Все более актуальным становится вопрос научно-обоснованной разработки отказоустойчивых систем управления. Важность повышения надежности программного обеспечения обусловлена тем, что оно выполняет основные функции системного управления обработкой данных, и его отказы в работе могут оказать существенное влияние на функционирование систем обработки данных и управления в целом.

Создание отказоустойчивого ПО невозможно без средств анализа надежности. Если не проводить анализ надежности на ранних этапах разработки, возможна ситуация, когда уже разработанный программный комплекс при функционировании не обеспечивает заданных параметров надежности СУ, что приводит к необходимости повторной разработки ПО с использованием других моделей и алгоритмов. Это обстоятельство создает риск существенного удорожания и увеличения времени разработки. Возможность осуществить анализ надежности ПО на более ранних этапах проектирования системы делает ее разработку более предсказуемой и эффективной.

Таким образом, создание модельно-алгоритмического обеспечения анализа отказоустойчивости программных комплексов встраиваемых систем управления реального времени является актуальным.

Целью настоящего диссертационного исследования является повышение отказоустойчивости разрабатываемого программного обеспечения за счет получения оценок надежности программных модулей на этапе проектирования и алгоритмов принятия решения в мультиверсионных системах.

Для достижения поставленной цели решались следующие задачи.

- анализ существующих моделей отказоустойчивых систем управления, механизмов обеспечения надежности СУ с применением программной избыточности;

- разработка типовой структуры мультиверсионной СУ на основе операционной системы реального времени (ОСРВ);

- разработка комбинированного селективного алгоритма выбора мультиверсионной модели повышения отказоустойчивости для реализации программного комплекса СУ с требуемыми характеристиками надежности. Алгоритм основан на модифицированной модели деревьев сбоев и впервые предложенной модели расчета величины корректности;

- разработка новых и модификация существующих алгоритмов принятия решения в мультиверсионных системах, позволяющих повысить отказоустойчивость;

- реализация имитационной среды моделирования (анализа надежностных характеристик) мультиверсионного программного комплекса, для возможности тестирования и получения характеристик исследуемых моделей, предложенных модифицированных алгоритмов голосования, возможности их сравнения в одинаковых условиях с классическими алгоритмами;

- реализация мультиверсионной среды исполнения реального времени для валидации предложенных моделей и алгоритмов;

- разработка модели прогнозирования надежности на основе результатов тестирования программных модулей.

Область исследования. Работа выполнена в соответствии со следующими пунктами паспорта специальности 05.13.01:

- разработка специального математического и алгоритмического обеспечения систем анализа, оптимизации, управления, принятия решений и обработки информации;

- методы и алгоритмы прогнозирования и оценки эффективности, качества и надежности сложных систем.

Методы исследования.

При выполнении работы использовались методы и подходы теории вероятности, имитационного моделирования, методы анализа и проектирования информационных систем, системного анализа, теории надежности, мультиверсионного проектирования программного обеспечения отказоустойчивых систем управления, методы теории графов.

Научная новизна работы.

1. Впервые сформирована типовая структура мультиверсионной системы управления реального времени, в том числе ее программная составляющая, которая позволяет: разработать имитационную среду моделирования, в которой тестируются отдельные программные модули, определяются их характеристики и общие характеристики системы при их применении; реализовать мультиверсионную среду исполнения реального времени на основе ОСРВ, применимую на большинстве современных одноплатных компьютеров.

2. Предложен и реализован комбинированный селективный алгоритм оценки эффективности мультиверсионных моделей, впервые позволивший получить верхнюю и нижнюю границы надежности. Нижняя граница рассчитывается с применением модели «деревьев сбоев», верхняя – с применением модели корректности мультиверсионной системы. Оригинальность данных подходов заключается в расчете характеристик надежности разрабатываемой системы, с учетом характеристик составляющих ее программных и аппаратных модулей, и структуры системы (используя знания о структуре системы, в зависимости от применяемых в ней мультиверсионных моделей).

3. Предложены и реализованы модификации алгоритмов голосования согласованным большинством и нечеткого голосования согласованным большинством. Для повышения устойчивости алгоритмов к межверсионным ошибкам программных модулей, по сравнению с невзвешенными аналогами, введены веса версий и элемент забывания.

4. Разработан новый алгоритм предсказания времени наработки на отказ на основе данных автоматизированного тестирования. Алгоритм основан на экспоненциальном распределении времени нахождения ошибок в программной системе и на основании данных об уже выявленных ошибках позволяет предсказать время работы системы до проявления следующей ошибки, что и является временем наработки на отказ. Алгоритм позволяет узнать предполагаемый диапазон значения времени наработки на отказ и вероятность попадания в предсказанный диапазон.

Значение для теории состоит в разработке новых взвешенных алгоритмов голосования с забыванием, методик предсказания времени наработки на отказ, новых методов оценки надежности мультиверсионных систем. Результаты, полученные при выполнении диссертационной работы, создают теоретическую основу для разработки новых технологий проектирования мультиверсионного программного обеспечения сложных систем управления. Благодаря предложенным методам и алгоритмам обеспечивается повышение эффективности процессов обработки информации и управления.

Практическая ценность.

Предложенная в диссертации технология разработки отказоустойчивых программных комплексов систем управления позволяет автоматизировать процесс решения задачи компоновки состава мультиверсионного программного комплекса, выбрать алгоритмы принятия решения, сформировать требования к надежности составляющих систему программных блоков. Применения технологии позволяет оценить характеристики системы на ранних этапах разработки, что не только позволит выбрать оптимальные алгоритмы и программные модули, но и существенно уменьшит риск реализации системы, которая на практике не будет удовлетворять поставленным требованиям надежности.

Типовая структура отказоустойчивого программного комплекса позволит создавать мультиверсионную среду исполнения на базе ОСРВ FreeRTOS, применимую на большинстве современных одноплатных компьютеров и платах разработчиков.

Применение модифицированных алгоритмов голосования не только увеличивает устойчивость мультиверсионных систем к межверсионным ошибкам, но и способствует увеличению надежности системы в целом.

Применение методики предсказания времени наработки на отказ существенно сокращает время на этапе тестирования, поскольку дает возможность тестировать систему не в масштабе реального времени, а настолько быстрее, насколько производительна тестовая платформа, что позволяет гарантировать время

наработки на отказ в месяцах, затрачивая на тестирование на порядок меньшие промежутки времени.

Реализованная имитационная среда моделирования мультиверсионных программных комплексов позволяет тестировать и получать характеристики исследуемых моделей, предложенных модифицированных алгоритмов голосования, а также дает возможность их сравнения в одинаковых условиях с классическими алгоритмами. Оригинальность данной методики заключается в сравнении моделей и алгоритмов с помощью анализа их реальной работы в рамках имитационной среды моделирования, которая симулирует выходы версий с заданными характеристиками и собирает статистику работы всех алгоритмов на каждой итерации.

Достоверность полученных результатов подтверждается результатами моделирования в имитационной среде и реализованной мультиверсионной средой исполнения реального времени, проведенными тестами системы на реальной задаче. Непротиворечивостью использованных моделей, методов и численных результатов моделирования и расчетов.

Реализация результатов работы. В диссертационной работе были разработаны шесть программных систем, четыре из них прошли регистрацию в Роспатенте.

Диссертационная работа выполнена в рамках проектов:

РФФИ № 16-47-242143 «Мультиверсионная среда исполнения алгоритмов управления автономными беспилотными объектами»;

Госзадание № 2.2867.2017/ПЧ «Технология организации жизненного цикла кроссплатформенного программного обеспечения бортовой аппаратуры беспилотных летательных объектов».

Апробация работы. Основные положения и результаты работы прошли всестороннюю апробацию на международных конференциях.

1. Международная научно-практическая конференция «Современное состояние науки и техники» ССНиТ, Сочи, 2016 г.

2. XIX Международная научно-практическая конференция, посвященная 55-летию Сибирского государственного аэрокосмического университета имени академика М. Ф. Решетнева “Решетнёвские чтения”, Красноярск, СибГАУ, 2015 г.

3. XVIII Международная научная конференция, посвященная 90-летию со дня рождения генерального конструктора ракетно-космических систем академика М. Ф. Решетнева “Решетнёвские чтения”, Красноярск, СибГАУ, 2014 г.

4. Международная научно-практическая конференция «Теоретические и прикладные проблемы науки и образования в 21 веке», Тамбов, 2012 г.

5. Международная научная конференция «Современные проблемы математического моделирования, обработки изображений и параллельных вычислений 2017», пос. Дивноморское, Геленджик, 2017 г.

6. XIII Международная научно-практическая конференция, посвященная дню космонавтики «АКТУАЛЬНЫЕ ПРОБЛЕМЫ АВИАЦИИ И КОСМОНАВТИКИ», Красноярск, 2017 г.

7. XIII Международная научно-техническая конференция «Динамика технических систем» (ДТС-2017), Ростов-на-Дону, 2017 г.

8. European Conference on Electrical Engineering and Computer Science (EECS 2017), Берн, Швейцария, 2017 г.

9. INTE 2017 International Conference on New Horizons in Education, Берлин, Германия, 2017 г.

Публикации. По теме диссертации опубликовано 19 печатных работ, из них пять статей в журналах перечня ВАК РФ и шесть в изданиях, индексируемых в международных базах цитирования Web of Science и/или Scopus. Получено четыре свидетельства о регистрации программных систем в Роспатенте. Полный список публикаций представлен в конце автореферата.

Структура и объем диссертации. Диссертация состоит из введения, пяти глав, заключения, списка использованной литературы и приложения. Она изложена на 203 листах машинописного текста, содержит список литературы из 153 наименований.

1 Программный комплекс критичных по надежности встраиваемых систем управления реального времени

Проектирование алгоритмов систем управления в том числе автономными беспилотными объектами (АБО), не является тривиальной задачей, поскольку система управления является сложной системой, элементы которой могут быть интерпретированы как логической, так и физической архитектурой.

1.1 Отказоустойчивые системы управления для различных классов АБО

На текущий момент актуальные системы управления разрабатываются на основе универсальных или специализированных аппаратных платформ, при этом возросшие вычислительные мощности и коммуникационные возможности подобных платформ создают существенную сложность объекта и процесса управления. Это также определяет и сложность ПО для управления данными объектами и комплексами объектов – систем управления.

В связи с этим ПО может быть рассмотрено как сложная система.

Наиболее существенные признаки сложных систем [1]:

- существование глобальной задачи и цели, для выполнения которой работают все элементы структуры системы;
- большое количество структурных единиц, на разных уровнях иерархии, которые взаимодействуют друг с другом;
- возможность разбиения на подсистемы – наборы элементов, наиболее тесно взаимодействующих друг с другом, имеющие сходную подзадачу и цель функционирования;
- структура связей подсистем с четкой иерархией и формализованные критерии качества функционирования всей системы;
- априорная неопределенность поведения системы и вытекающая из этого сложность, связанная со случайным характером внешних воздействий и большим количеством обратных связей внутри системы;

- наличие механизмов, обеспечивающих устойчивость по отношению к внешним и внутренним сбоям и наличие самоорганизации и адаптации к различного вида возмущениям;
- высокая общая надежность системы, построенной из не всегда абсолютно надежных компонентов.

Разработка систем управления, относящихся к классу сложных, с требуемыми характеристиками при ресурсных и временным ограничениях, требует проведения определенного комплекса мероприятий для достижения поставленных целей.

Функционирование программного обеспечения в составе сложных систем управления имеет следующие основные особенности.

Работа в реальном времени обусловлена критичностью ко времени реакции системы управления, верный результат вычисления, но не своевременный, может привести к серьезным негативным последствиям, вплоть до разрушения объекта. Поэтому столь критично не только обеспечить отсутствие программных сбоев, но и гарантировать скорость получения реакции системы. Одной из важнейших задач управления является организация множества вычислений в реальном времени.

Разнообразие функций обусловлено тем, что в рамках системы управления одновременно выполняется множество совершенно разных задач – от управления моторами и процесса приема и передачи данных до процесса распознавания и классификации образов, мультиверсионного голосования и т.д.

При функционировании системы осуществляется асинхронное исполнение множества задач и информационных транзакций.

Надежность функционирования системы при возникновении программных и аппаратных сбоев и отказах, которые неизбежно могут возникать при эксплуатации системы. Причинами могут служить как заложенный производителем брак в случае аппаратного обеспечения или допущенные алгоритмические или синтаксические ошибки в коде в случае программного обеспечения, так и особенности эксплуатации готовой системы. Для повышения надежности в подобных системах находят применение различные методики

контроля, самопроверки, введение как аппаратной, так и программной избыточности, инструменты восстановления после сбоев.

Гарантированное время наработки на отказ – система должна иметь гарантированный срок, в течении которого не возникнет отказа, то есть система должна адекватно проработать на протяжении указанного отрезка времени.

Классификация автономных беспилотных объектов

С развитием робототехники в последние годы все более актуальными становятся автономные беспилотные объекты. Причиной этому служит как развитие технической базы - появление компактных контроллеров с все более высокой вычислительной мощностью, точных датчиков, камер с высокой разрешающей способностью, появление рынка готовых изделий, например - круговой лазерный сканер (лидар) может быть свободно приобретен в розничной продаже, это потребительский товар и на рынке уже представлен не один производитель. Также с каждым годом электронные компоненты становятся все компактнее и энергоэффективнее - потребляют меньше электрической мощности, выделяют меньше тепла, благодаря переходу производителей полупроводниковой продукции на более тонкий техпроцесс и совершенствованию микроархитектуры чипов.

Вторым фактором является наработка алгоритмической базы, автономное управление уже не первый год является актуальной задачей и это направление активно развивается, появляются новые алгоритмы и методы, готовые программные решения. Что можно назвать автономным беспилотным объектом (АБО)? Это объект, способный перемещаться (будь то подводные, наводные, наземные, воздушные объекты) или корректировать свое позиционирование в пространстве (космические объекты) на основании некоторого набора входных данных (изображение с камер, сканирующих лидаров, показания различных датчиков, заранее заданный маршрут с привязкой к местности) и способный либо выполнять манипуляции с окружающими объектами, либо собирать/отправлять данные. У подобных объектов может иметься возможность перехвата управления вручную, по каналу связи, однако это не обязательно и не всегда удобно либо вовсе

не реализуемо (например, работа в среде, в которой невозможно прохождение радиосигналов и обеспечение иных способов реализации канала связи), определяющая характеристика - способность полностью автономной работы, без участия оператора или поступления иных команд из вне.

Условно АБО можно разделить на следующие классы [2]:

Наземные – группа объектов, предназначенных для передвижения по поверхности земли, городским улицам, дорогам, внутри зданий и т.д. Это самый распространенный класс, включающий в себя множество различных по назначению АБО, от учебных платформ, на которых школьники и студенты познают азы робототехнического конструирования и программирования контроллеров до специализированных военных роботов-саперов и строительных роботов, участвующих в ликвидации аварий на атомных электростанциях.

Для учебных целей данный класс наиболее прост и безопасен, так как позволяет проводить занятия в обычной аудитории и даже собирать прототипы у себя дома, а в случае отказа каких-либо блоков не представляет особой опасности как с точки зрения возможной аварии при взаимодействии с окружающими объектами так и с точки зрения потери самого АБО. На учебных платформах как правило установлены маломощные мотор-редукторы, сами платформы имеют незначительный вес и кнопку аварийного отключения, либо всего АБО, включая электронные системы, либо кнопку отключения питания моторов или иных механизмов, приводящих его в движение, что также бывает полезно при отладке, когда исследуется реакция управляющих систем на информацию, поступающую с датчиков.

Для практических целей данный класс объектов находит применение там, где либо не может проникнуть человек, например, небольшой АБО может проникать между обломков при разборе последствий разрушения здания, либо при работе в местах, представляющих опасность для человека, таких как места активных вооруженных конфликтов для военных роботов, минных полей или мест с потенциальным нахождением взрывных устройств для роботов-саперов, мест радиационного заражения для строительных и специализированных роботов.

Последние годы этот класс АБО находит применение и в бизнесе, уже существуют целые склады, работающие в автоматическом режиме, значительно ускоряя решение логистических задач и повышая эффективность работы, в данном случае, как правило, имеется одна центральная система управления - информационная система, принимающая решение о логистических операциях, исходя из поступающей информации о заказах и перемещениях, и множество «исполнителей», физически осуществляющих данные операции, однако, все работы происходят автономно, без участия человека.

Водные – этот класс можно условно разбить на наводные и подводные объекты, что интересно - несмотря на значительно более сложную техническую реализацию именно подводных решений, связанных с герметичностью, противостоянию высокому давлению, зачастую экстремальному (в зависимости от глубины погружения), прохождением сигнала в воде, низкой видимостью и т.д., они получили наибольшее распространение, так как решают круг задач, где наиболее востребована «беспилотность».

В военной отрасли подводные АБО могут выполнять как задачи разведки и обнаружения, так и боевые задачи, доставляя на себе оружие, благодаря небольшим размерам, малошумности, возможности медленного передвижения и отсутствию теплового следа их крайне сложно обнаружить, а в автономном режиме нет и радиосвязи, которая могла бы стать причиной обнаружения объекта.

В гражданских отраслях подводные АБО служат для обследования кабелей, проложенных по дну водоемов, обследованию затонувшей техники, подводных работ, научных исследований. Они позволяют существенно снизить стоимость оборудования, поскольку к подводному устройству, не предназначенному для нахождения в нем человека применяются другие требования, нет необходимости в особой атмосфере внутри, азота и кислорода для дыхания, габариты без пилота внутри существенно уменьшаются, а сделать маленький корпус устойчивым к высокому давлению проще и дешевле. Что также очень важно, уходит риск для человека при выполнении подобных работ, поскольку они либо проходят в

автономном режиме, либо под управлением оператора, находящегося на поверхности воды, вне опасности.

Воздушные - к этому классу относятся как классические крылатые беспилотники и ракетная техника, так и набирающие популярность мультироторные системы (квадро-, гекса-, октокоптеры). В случае воздушных полетов вес и габариты крайне важны, поскольку поднять в небо среднего человека уже существенная задача, требующая совершенно иного уровня аппаратных средств, по сравнению с поднятием зачастую некрупной полезной нагрузки, например - оборудования для видео или фотосъемки.

Отсутствие пилота позволяет кардинально снизить вес и уменьшить габариты летательного аппарата, соответственно снижаются и затраты на производство и сами полеты, в большинстве случаев отпадает необходимость в топливных двигателях, достаточно электромоторов, работающих от аккумуляторов. Небольшие габариты и отсутствие топливных двигателей - источников большого количества тепла существенно усложняют обнаружение подобных летательных средств.

Основным недостатком беспилотных летательных аппаратов считается уязвимость радиоканала связи для удаленного управления, автономный режим решает эту проблему, поскольку даже полная потеря канала не выводит аппарат из строя, он продолжает выполнять задачу. Отсутствие радиоканала также является существенным преимуществом в плане сложности обнаружения объекта. На данный момент воздушные АБО применяются как в военной отрасли – от самолетов-разведчиков до боевой техники, несущей на себе вооружение, так и в повседневном использовании – уже практикуется доставка потребительских товаров с помощью автономных беспилотных мультироторных систем.

Специализированные – АБО предназначенные для узкоспециализированных работ в определенных условиях, например – передвижение, исследование и работы внутри магистральных газопроводов, иных коммуникаций, отсеки танкеров и т.д.

Нахождение человека в подобных средах не всегда возможно или крайне опасно. Необходимость автономной работы вызвана невозможностью реализации

канала связи, например - многокилометровые металлические трубопроводы, имеющие заземление и экранирующие объект от радиосигналов из вне, а кабельная связь в подобных условиях нецелесообразна или не осуществима.

Отнести данный класс объектов к основным классам сложно, поскольку они созданы для работы в строго определенных условиях, например, способны передвигаться только по трубопроводам заданного диаметра в известной среде. В зависимости от условий, для которых они создавались, АБО обладают такими конструктивными особенностями, как герметичный корпус, решения, предотвращающие накопление статического электричества и искрообразование при работе в среде, наполненной воспламеняемым газом и т.д.

Космические – к данному классу относятся такие решения для постоянного нахождения в открытом космосе, как спутники на околоземной орбите, так и аппараты, запускаемые на другие космические тела, предназначенные для передвижения по ним и активных действий – сбор и анализ образцов поверхности, атмосферы, фото, видеосъемка с дальнейшей отправкой результатов на землю.

Несмотря на реализацию радиоканала связи, управление подобными объектами крайне затруднительно, поскольку на космических расстояниях начинает иметь значение скорость радиосигнала, внося существенные задержки как в сторону передачи управляющих команд, так и в сторону отправки видео или иного сигнала с управляемого объекта.

На сегодняшний день дальние космические полеты осуществимы только беспилотными аппаратами, поскольку отправить человека на расстояние нескольких световых лет и вернуть его назад практически неосуществимо на данном этапе развития технологий. Наличие человека для подобных миссий не требуется, а его отсутствие позволяет сделать космический аппарат существенно легче, компактнее и проще, что очень важно для космических запусков.

Благодаря миниатюризации электронных компонентов стали возможны такие проекты, как кубсат (англ. CubeSAT) и покетсат (англ. PocketSAT), дающие шанс на реальный запуск в космос студенческих малых спутников. Студенты создают свои собственные спутники, начиняя их различной электроникой на свое

усмотрение, ограничиваясь только заданными стандартом габаритами корпуса, формой крепежных элементов и весом, потом готовые спутники запускаются на околоземную орбиту и студенты с преподавателями могут отправлять или получать с него сигналы в университетских ЦУПах или иными путями, в зависимости от электронных компонентов в разработанном спутнике.

Автономные беспилотные объекты уже проникают во многие сферы жизни современного человека, в дальнейшем их распространение и количество решаемых задач будет только увеличиваться. Разработка технических решений и программного обеспечения для них является актуальной задачей, как для дальнейшего развития существующих АБО, так и для создания принципиально новых, для тех областей, где ранее они не применялись.

Системы управления для подобных объектов должны удовлетворять не только требованиям отказоустойчивости, но и быть применимыми на различных аппаратных платформах, поскольку в зависимости от класса АБО и области применения в них используются различные контроллеры и аппаратные решения. Идеальной, в данном случае, будет единая программная система управления, разработанная и отлаженная с учетом всех требований надежности, поддерживающая работу в режиме реального времени, реализующая механизмы повышения отказоустойчивости (программная избыточность, алгоритмы голосования, приемочные тесты и т.д.), которую возможно применять на разных аппаратных платформах, в зависимости от типа и области применения АБО [3].

Для разработки подобных систем и наделения их требуемыми свойствами нам необходимо учитывать особые требования к отказоустойчивости на каждом этапе разработки, для этого необходимо понимать последовательность разработки программных продуктов.

Поскольку в качестве объекта управления мы рассматриваем АБО, то можно конкретизировать интересующий нас класс систем управления. Для управления АБО критично время отклика, СУ должна работать в режиме реального времени. Так же СУ должна располагаться непосредственно на объекте, для обеспечения

возможности автономной работы и уменьшения задержек. Таким образом, будут рассматриваться встраиваемые системы управления реального времени.

1.2 Анализ жизненного цикла разработки программных комплексов критичных по надежности встраиваемых систем управления реального времени

Жизненный цикл программного обеспечения включает в себя все этапы развития: от возникновения потребности в нем до полного прекращения его использования вследствие потери необходимости решения соответствующих задач или окончания срока службы устройств, на которых функционировала разработанная система.

Можно выделить несколько этапов существования программного изделия в течение его жизненного цикла. На текущий момент нет общепринятого разбиения и наименования этих этапов, поскольку они могут различаться в зависимости от назначений, масштаба и области применения программного обеспечения.

По длительности жизненного цикла программные изделия можно разделить на два класса: с малым и большим временем жизни. Этим классам программ соответствуют более гибкий подход к их созданию и эксплуатации и жесткий промышленный подход проектирования и эксплуатации программного обеспечения. Например, в научных организациях и вузах, преобладают разработки первого класса, а в проектных и промышленных организациях - второго.

Программное обеспечение с малой длительностью эксплуатации

Данное программное обеспечение создается в основном для решения научных и инженерных задач, для получения конкретных результатов вычислений, выходов исследуемых моделей, выявления зависимостей. Такие программы, как правило, относительно невелики. Они разрабатываются одним специалистом или небольшой группой.

Их жизненный цикл состоит из длительного интервала системного анализа и формализации проблемы, значительного этапа проектирования и относительно небольшого времени эксплуатации и получения результатов, что иллюстрирует

рисунок 1. Требования, предъявляемые к функциональным и конструктивным характеристикам, как правило, не формализуются, отсутствуют оформленные испытания программ. Показатели их качества контролируются только разработчиками в соответствии с их неформальными представлениями.

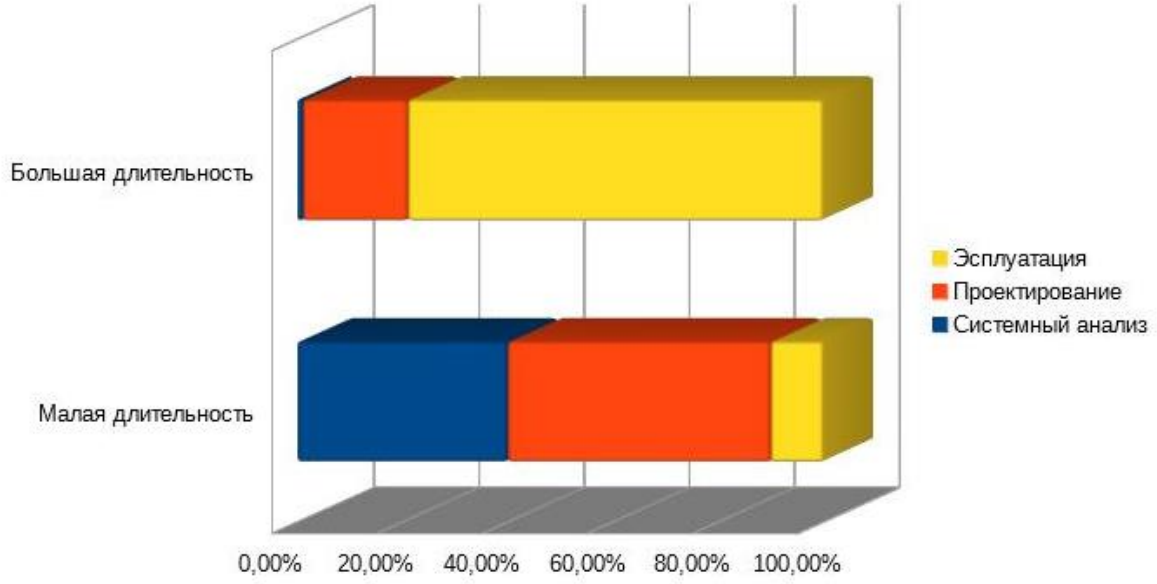


Рисунок 1 - Программные изделия с малой и большой длительностью эксплуатации

Сопровождение и модификация таких программ не обязательны, и их жизненный цикл завершается после получения результатов вычислений.

Программное обеспечение с большой длительностью эксплуатации

Данное программное обеспечение создается для регулярной обработки информации и управления. Структура таких программ довольно сложна. Их размеры могут изменяться в широких пределах от тысячи до миллионов команд, однако все они обладают свойствами познаваемости и возможности модификации в процессе длительного сопровождения и использования различными специалистами. Программные комплексы этого класса допускают тиражирование, они сопровождаются документацией как промышленные изделия и представляют собой отчуждаемые от разработчика программные продукты.

Их разработкой и поддержкой в период эксплуатации занимаются крупные коллективы специалистов, для чего необходима формализация программной системы, а также формализованные испытания и определение достигнутых показателей качества конечного продукта.

Поскольку системы управления являются программными изделиями с большой длительностью эксплуатации, сосредоточимся на них.

Обобщенная модель *жизненного цикла* программного изделия может выглядеть так:

I. На этапе *системного анализа проводят:*

1.1 исследования;

1.2 анализ осуществимости:

- эксплуатационной;
- экономической;
- коммерческой.

II. *На этапе проектирования программного обеспечения:*

2.1 конструирование:

- функциональная декомпозиция системы, ее архитектура;
- внешнее проектирование программного обеспечения;
- проектирование базы данных;
- архитектура программного обеспечения.

2.2 программирование:

- внутреннее проектирование программного обеспечения;
- внешнее проектирование программных модулей;
- внутреннее проектирование программных модулей;
- кодирование;
- отладка программ;
- компоновка программ.

2.3 отладка программного обеспечения.

III. *Этап оценки (испытания) программного обеспечения.*

IV. *На этапе использования программного обеспечения происходит:*

4.1 эксплуатация;

4.2 сопровождение.

Этап системного анализа. В начале разработки программного обеспечения проводят системный анализ, в ходе которого определяются потребность в нем, назначение и основные функциональные характеристики. Оцениваются затраты и возможная эффективность применения будущего программного обеспечения. В случае разработки критичных по надежности систем, оцениваются необходимые характеристики надежности.

На этом этапе составляется перечень требований, то есть четкое определение того, что ожидается от программного продукта. Здесь же осуществляется постановка целей и задач. На этапе системного анализа можно выделить два направления: исследование и анализ осуществимости.

Исследования начинаются с того момента, когда возникает осознанная потребность в программном обеспечении.

Работа состоит в планировании и координации действий, необходимых для подготовки формального рукописного перечня требований к разрабатываемому программному обеспечению.

Исследования заканчиваются тогда, когда требования формализованы и, при необходимости, могут быть модифицированы и одобрены ответственным руководителем.

Анализ осуществимости - это техническая часть исследований и начинается тогда, когда назначается руководитель проекта, организующий проектирование и распределение ресурсов.

Работа заключается в исследовании предполагаемого программного обеспечения с целью получения практической оценки возможности реализации проекта, в частности определяются:

- *осуществимость эксплуатационная,* будет ли изделие удобным для практического использования, реализуемы ли все функциональные задачи в рамках

целевой аппаратной платформы, достижимы ли требуемые характеристики надежности.

- *осуществимость экономическая,* приемлема ли стоимость разрабатываемого изделия? Какова эта стоимость? Может рассматриваться ряд вариантов разработки, с прогнозируемыми эксплуатационными характеристиками и стоимостью разработки, принимается решение о возможной разработке прототипа, для более обоснованного выбора модели разработки, если это необходимо.

- *осуществимость коммерческая,* будет ли изделие привлекательным, пользоваться спросом? Этот пункт необходим только в случае разработки систем управления, нацеленных на дальнейшую коммерческую реализацию.

Анализ осуществимости заканчивается, когда все требования собраны и одобрены.

Прежде чем продолжить дальнейшую разработку, необходимо объединить все формализованные требования в спецификацию, отсутствие которой существенно затруднит дальнейшие работы.

Часто в период системного анализа принимается решение о прекращении дальнейшей разработки программного обеспечения.

Проектирование программного обеспечения. Проектирование является основным и решающим этапом жизненного цикла программного обеспечения, во время которого создается и на 90% приобретает свою окончательную форму программное обеспечение.

Этот этап жизненного цикла охватывает различные виды деятельности проекта и может быть разделен на три основных фазы: конструирование, программирование и отладку программного обеспечения.

Конструирование программного обеспечения обычно начинается ещё на этапе анализа осуществимости, как только оказываются зафиксированными на бумаге некоторые предварительные цели и требования.

На этом отрезке жизни программного обеспечения осуществляют:

- функциональную декомпозицию решаемой задачи, на основе которой определяется архитектура системы этой задачи;

- внешнее проектирование программного обеспечения, выражающееся в форме внешнего взаимодействия его с пользователем;

- проектирование базы данных, если это необходимо;

- проектирование архитектуры программного обеспечения — определение объектов, модулей и их сопряжения.

На данном этапе выбираются подходящие модели повышения отказоустойчивости программного обеспечения (моно- или мультиверсионные), которые позволят достичь требуемых характеристик надежности, при необходимости, закладывается избыточность.

Программирование начинается уже в фазе конструирования, как только станут доступными основные спецификации на отдельные компоненты программного изделия, но не ранее утверждения соглашения о требованиях. Перекрытие фаз программирования и конструирования приводит к экономии общего времени разработки, а также к обеспечению проверки правильности проектных решений, и в некоторых случаях влияет на решение ключевых вопросов.

На этом этапе выполняется работа, связанная со сборкой программного изделия. Она состоит во внутреннем конструировании программного продукта, в разработке внутренней логики каждого модуля системы, которая затем выражается в виде кода конкретной программы.

На данном этапе важно соблюдать требования выбранной ранее модели повышения отказоустойчивости, например, в случае выбора N-версионного программирования, различные версии должны разрабатываться разными людьми, с использованием различных инструментов разработки, компиляторов, возможно на разных языках программирования, чтобы минимизировать межверсионные ошибки [4].

Этап программирования завершается, когда разработчики закончат документирование, отладку и компоновку отдельных частей программного изделия в одно целое. В случае особых требований, к примеру – по

отказоустойчивости, к определенным компонентам системы, при их отладке сначала проверяются характеристики отдельного блока, однако окончательные выводы о соответствии заявленным характеристикам делаются только на этапе отладки собранной воедино системы, поскольку межмодульное взаимодействие может повлиять на характеристики отдельных программных блоков.

Отладка программного обеспечения осуществляется после того, когда все его компоненты будут отлажены по отдельности и собраны в единый программный продукт.

На данном этапе применяются модели надежности программного обеспечения, которые определяются в зависимости от известных входных данных и целевых показателей выходных данных модели, которые необходимо достичь. К примеру, не всегда применимы модели, требующие знания всех параметров программной системы, ибо даже при самостоятельной разработке в ней могут быть использованы сторонние закрытые модули и библиотеки, внутреннее строение которых неизвестно. В этих случаях применяются модели Джелинского-Моранды и другие, не требующие полной информации о строении всей отлаживаемой программы. Исправляются все выявленные при отладке ошибки.

Оценка (испытания) программного обеспечения. На данном этапе программное обеспечение подвергается строгому системному испытанию со стороны специалистов, которые не являются разработчиками данного продукта.

Это делается для того, чтобы гарантировать, что готовое программное обеспечение удовлетворяет всем требованиям и спецификациям, может быть использовано в среде пользователя, свободно от каких-либо дефектов и содержит необходимую документацию.

Фаза оценки начинается, как только все компоненты (модули) собраны вместе и испытаны, т.е. после полной отладки готового программного продукта. Она заканчивается после получения подтверждения требуемых характеристик системы.

Данный этап может занимать до 60% от общего времени создания программного обеспечения.

Использование программного обеспечения. Этап использования программного обеспечения начинается тогда, когда оно запускается на целевой аппаратной платформе и начинает работать по назначению.

Это время, на протяжении которого программное обеспечение находится в эксплуатации и используется эффективно.

На данном этапе выполняются обучение персонала, внедрение, настройка, сопровождение и, возможно, расширение программного продукта.

Этап использования заканчивается, когда программное обеспечение изымается из употребления и упомянутые выше действия прекращаются. Использование программного продукта определяется его эксплуатацией и сопровождением.

Эксплуатация программного обеспечения заключается в исполнении, функционировании его на целевой аппаратной платформе и в получении результатов, являющихся целью его создания.

Сопровождение программного обеспечения состоит в эксплуатационном обслуживании, развитии функциональных возможностей и повышении эксплуатационных характеристик программного обеспечения.

Сопровождение играет роль необходимой обратной связи от этапа эксплуатации.

Эти доработки, как правило, ведутся одновременно с эксплуатацией текущей версии программного обеспечения. Эти обстоятельства приводят к тому, что процесс эксплуатации версии программного изделия обычно идет параллельно и независимо от этапа сопровождения.

Рисунок 2 - Перекрытие между фазами жизненного цикла программного изделия

Возможны и обычно желательны перекрытия между разными фазами жизненного цикла программного изделия (рисунок 2). Однако не должно быть никакого перекрытия между несмежными процессами.

Возможна обратная связь между фазами. Например, во время одного из шагов внешнего проектирования могут быть обнаружены погрешности в формулировке целей, в таком случае следует вернуться и исправить их.

1.3 Модели надежности программного обеспечения

Таким образом, при решении задачи проектирования, тестирования и эксплуатации существует потребность в модельно-алгоритмическом сопровождении надежности программного обеспечения.

Термин *модель надежности программного обеспечения*, как правило, относится к математической модели, построенной для оценки зависимости надежности программного обеспечения от некоторых определенных параметров. Значения таких параметров либо предполагаются известными, либо могут быть измерены в ходе наблюдений или экспериментального исследования процесса функционирования программного обеспечения. Данный термин может быть использован так же применительно к математической зависимости между определенными параметрами, которые хотя и имеют отношение к оценке надежности программного обеспечения, но тем не менее не содержат ее характеристик в явном виде. Таким параметром является частота ошибок, которая позволяет оценить именно качество систем реального времени, функционирующих в непрерывном режиме, и в то же время получать только косвенную информацию относительно надежности программного обеспечения [5].

Рассмотрим некоторые модели оценки и прогнозирования надежности программного обеспечения, как потенциально возможные к применению при разработке системы управления АБО.

Модель Шумана. Целесообразность применения этой модели для оценки надежности программного обеспечения зависит от принятых допущений и условий, наиболее существенным из которых является *условие существования*

«программы исследователя системы». Остальные допущения и условия носят статический характер и не связаны с какими-либо специфическими свойствами программного обеспечения. По существу, они сводятся к следующему.

- Предполагается, что в начальный момент компоновки программных средств в систему программного обеспечения в них имеется ЕТ ошибок. С этого момента начинается отчет времени отладки τ, которое включает затраты времени на выявление ошибок с помощью тестов, на контрольные проверки и т.п., при этом время исправного функционирования системы не учитывается.

В течение времени отладки τ устраняется $\varepsilon_c(\tau)$ ошибок в расчет на одну команду в машинном языке. Таким образом, удельное число ошибок на одну машинную команду, остающихся в системе после τ месяцев отладки, равно:

$$\varepsilon_r(\tau) = \frac{E_T}{I_T} - \varepsilon_c(\tau), \tag{1}$$

где I_T – общее число машинных команд, которое предполагается постоянным.

- Предполагается, что значение функции частоты отказов $z(t)$ пропорционально числу ошибок, оставшихся в программном обеспечении после израсходования на отладку времени τ, т.е. $z(t) = C\varepsilon_r(\tau)$. Тогда, если число часов работы системы t отсчитывается от точки $t = 0$, а τ остается фиксированным, то функция надежности, или вероятность безотказной работы на интервале времени $(0, t)$, равна

$$R\ (t,\tau) = \exp\{-\ C\ [E_T\ /\ I_T - \varepsilon_c(\tau)]t\} \tag{2}$$

Если в ходе отладки прогоняются k тестов в интервалах $(0, \tau_1)$, $(0, \tau_2)$, $(0, \tau_k)$, где $\tau_1 < \tau_2 < \ldots < \tau_k$, то можно показать, что при $k \geq 2$ для определения оценок максимального правдоподобия $\hat{C}$ и $\hat{E}_T$ двух параметров C и E_T пригодны следующие уравнения:

$$\hat{\mathrm{C}} = \sum_{i=1}^{k} n_1 /' \sum_{i=1}^{k} \left[\widehat{E_t} /\ I_t - \varepsilon(\tau_i)\right] H_j$$

$$\hat{\mathrm{C}} = \left\{\sum_{j=1}^{k} n_j /[\widehat{E_T} /\ I_T - \varepsilon_C\ (\tau_j)]\right\} /' \sum_{j=1}^{k} H_j \tag{3}$$

где n_j – число прогонов j-го теста, оканчивающихся отказом; H_j – время, затраченное на выполнение успешных и безуспешных прогонов j – го теста.

Асимптотические значения дисперсий $\hat{C}$ и $\hat{E}_T$ (для больших значений n_j) даются следующими значениями:

$$\mathrm{Var}\,(\widehat{C}) \cong \frac{1}{\frac{\sum n_1}{C^2} - \frac{(\sum H_j)^2}{\sum \frac{n_j}{\varepsilon_r^2(\tau_j)}}}$$

$$\mathrm{Var}\,(\widehat{E_T}) \cong \frac{I_T^2}{\sum \frac{n_j}{\varepsilon_r^2(\tau_j)}} - \frac{c^2(\sum H_j)^2}{\sum n_j} \qquad (4)$$

Коэффициент корреляции определяется как

$$p(\hat{C}, \hat{E}_T) \cong \sum \frac{n_j}{\varepsilon_r(\tau_j)} /' \left[\sum n_j \sum \frac{n_j}{\varepsilon_r^2(\tau_j)}\right]^{1/2} \qquad (5)$$

Асимптотические значения дисперсий и коэффициента корреляции используются для определения доверительных интервалов значений параметров C и E_T на основе теории нормального распределения случайных величин.

Что же касается программы исследователя системы, то она должна снабжать испытуемые программные средства входными данными, отражающими реальные условия функционирования. Будем называть эти данные *функциональным разрезом* и определять их главным образом через распределение вероятностей значений входных переменных.

Модель Джелинского – Моранды. Одна из самых ранних моделей, которая все еще применяется сегодня, - это модель, разработанная Джелинским и Морандой при работе над некоторыми проектами по оборонному заказу.

Истекшее время между отказами берется за экспоненциальное распределение с параметром, пропорциональным числу оставшихся ошибок в программном обеспечении, т.е. среднее время между отказами в момент времени t равно *1/ф(N-(i - 1))*

Здесь t - это любой момент времени между возникновением *(i – 1)-го и* i-го случая сбоя. Величина представляет собой коэффициент пропорциональности, а N - общее количество ошибок в программном обеспечении с начального момента времени, в котором наблюдается программное обеспечение. Рисунок 3

иллюстрирует влияние обнаружения сбоев на уровень опасности. Можно заметить, что при обнаружении каждой ошибки коэффициент опасности снижается на константу пропорциональности **φ**. Это указывает на то, что воздействие каждого «устранения» сбоя одинаково. В классификационной схеме Мусы и Окумото это модель биномиального типа.

Предположения и требования к данным. Основные допущения:

1. Скорость обнаружения неисправности пропорциональна текущему уровню сбоя программного обеспечения.
2. Частота обнаружения неисправности остается постоянной в промежутках между возникновением неисправности.
3. Неисправность корректируется мгновенно без внесения новых ошибок в программное обеспечение.
4. Программное обеспечение работает аналогично тому, как это делается для прогнозирования надежности.
5. Каждая ошибка имеет одинаковые шансы быть обнаруженной в пределах класса точности, как и любая другая ошибка в этом классе.
6. Когда сбои (неисправности) обнаружены, отказы являются независимыми [6].

При этом, пронумерованные предположения с 4 по 6 являются достаточно стандартными, будем называть их стандартными предположениями для моделирования надежности. Предположение 4 состоит в том, чтобы гарантировать, что модельные оценки, полученные с использованием данных, собранных в одной конкретной среде, применимы к среде, в которой должны быть сделаны прогнозы надежности. Пятое предположение состоит в том, чтобы гарантировать, что различные отказы имеют одинаковые распределительные свойства. Один класс точности может иметь разную интенсивность отказов, чем другие, что требует отдельного анализа надежности. Последнее предположение позволяет упростить получение оценок максимального правдоподобия.

Требования к данным для реализации этой модели: истекшее время между отказами $X_1, x_2, \ldots, x_n$ или фактическое время, в течение которого программное обеспечение вышло из строя $t_1, t_2, \ldots, t_n$,где $x_i=t_i - t_{i-1}$, $i=1, \ldots, n$ *при* $t_0=0$

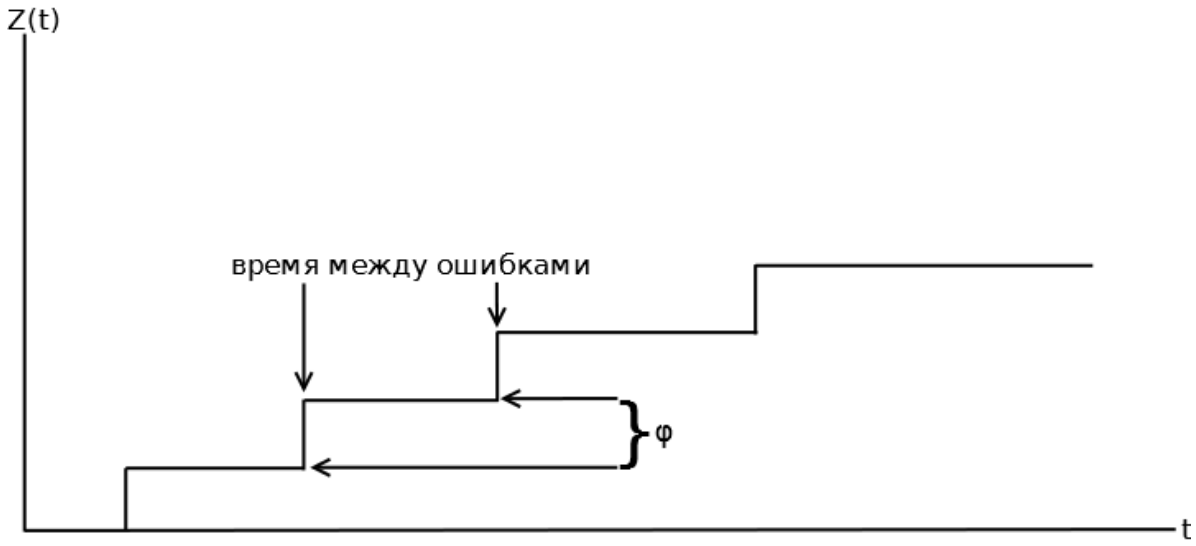


Рисунок 3 - Процесс выявления ошибок программного обеспечения.

Модельная форма. Из обзора модели и описанных ранее предположений, можем определить, что если промежутки времени между сбоями равны X_i=T_i-T_{i-1}, $i=1, \ldots, n$, *то* $X_i^{\cdot}S$ являются независимые экспоненциально распределенные случайные величины со средним = $1/\phi(N-(i-1)) = 1/z(X_i l\, T_{i-1})$. То есть

$$f(X_i l T_{i-1}) = z(X_i l T_{i-1})\exp(-z(X_i l T_{i-1})X_i)$$
$$= \phi[N-(i-1)]\exp(-\phi[N-(i-1])X_i \qquad (6)$$

Так как эта экспоненциальная модель принадлежит к биномиальному типу, то, используя формулы $\mu(t) = \alpha F_\alpha(t)$, где α некоторая константа, а $F_\alpha(t)$ - функция совокупного распределения времени до отказа отдельной неисправности α и $\lambda(t) = \mu'(t) = \alpha f_\alpha(t)$, где $f_a(t)$ функция плотности вероятности времени до отказа индивидуальной неисправности α, таким образом имеем: $\mu(t) = N(1-\exp(-\phi t))$ и $\lambda(t) = \mathrm{N}\phi\exp(-\phi t)$ для функции среднего значения и функции интенсивности отказов.

Очевидно, модель конечных отказов выглядит как

$$\lim_{t\to\infty}(\mu(t)) = \lim_{t\to\infty}\left(\mathrm{N}\left(1-\exp(-\phi t)\right)\right)$$
$$\widehat{\phi} = \frac{n}{\widehat{N}\ \left(\sum_{i=1}^{n} X_i\right) - \sum_{i=1}^{n}(i-1)X_i} \text{ и}$$
$$\sum_{i=1}^{n}\frac{1}{\widehat{N}-(i-1)} = \frac{n}{\widehat{N}-\left(1/\sum_{i=1}^{n}X_i\right)\left(\sum_{i=1}^{n}(i-1)X_i\right)} \qquad (7)$$

Второе уравнение решается численными методами для оценки максимального правдоподобия из N, и затем решение помещается в первое уравнение, чтобы найти оценку максимального правдоподобия ф.

Используя оценки максимального правдоподобия, различные меры надежности могут быть получены путем замены величин N и ϕ в интересующей функции надежности соответствующими оценками максимального правдоподобия $\hat{N}$ и $\hat{\phi}$.

Примером является оценочное значение MTTF (среднее время наработки на отказ) после обнаружения n неисправностей. Выражение для этого является MTTF для

(n + 1)-го сбоя = $1/z(x_{n+1}lt_n)$, поэтому по методу максимального правдоподобия (ММП) $MTTF = 1/\hat{z}(x_{n+1}lt_n) = 1/\widehat{\phi}(\widehat{N} - N)$.

Рассмотрим пример. Чтобы проиллюстрировать приведенные выше результаты, предположим, что мы наблюдали следующие события прошедшего времени между сбоями: $x_1 = 7, x_2 = 11, x_3 = 8, x_4 = 10, x_5 = 15, x_6 = 22, x_7 = 20, x_8 = 25, x_9 = 28, and\ x_{10} = 35.$

Используя полученные расчетные формулы получим оценки ММП для N и ϕ равны соответственно $\hat{N}$ = 11,6 и $\hat{\phi}$ = 0,0096. Таким образом, предполагаемая MTTF к следующей ошибке $MTTF = 1/\widehat{\phi}(\widehat{N} - n) = 1/0.0096(11.6 - 10) = 65.1.$

Модель Вейса. В этой модели помимо предположения об экспоненциальном распределении интервала времени между отказами считается, что существует M причин возможных отказов в начале процесса разработки программного обеспечения. Контрольные проверки, каждая длительностью T_t, осуществляются в произвольные моменты времени; фиксируются полученные результаты: время возникновения отказа или факт его отсутствия. При этом предполагается, что интенсивность отказов каждого типа различна и выражается некоторым априорным распределением вероятностей интенсивности отказов. Если при данном испытании происходит отказ, то вероятность того, что ошибка

корректируется, равна $p_c \leq 1$. Большинство следствий, вытекающих из этих предположений, демонстрируются на простых примерах. Основным результатом является вывод функции для предсказания надежности программного обеспечения, с помощью которой можно определить количество исключаемых ошибок при заданном числе испытаний.

Для дайной модели среднее время между отказами может быть аппроксимировано экспоненциальной функцией числа испытаний, для которой характерен постоянный процесс роста с увеличением числа испытаний.

Рассмотренная модель может быть полезной для оценки надежности программного обеспечения, если принять предположение, что каждая из M причин (источников) отказов представляет собой ошибки некоторого типа, многократно встречающиеся в программном обеспечении, например, «недостаточен выделенный объем памяти для массива данных».

Для ошибок некоторых типов, например, таких, как в описанном случае, существует высокая вероятность того, что при однократном их обнаружении и устранении возможность появления такой ошибки в будущем будет исключена. Для ошибок других типов однократное их обнаружение приводит к исключению подобных ошибок только из относительно небольшой части мест в программном обеспечении. Эта часть ошибок в описанной модели характеризуется вероятностью p_c, которая, вообще говоря, различна для каждого источника ошибок и должна быть предварительно оценена на основе фактических данных.

Как и во всех моделях для прогнозирования надежности функционирования технических устройств, в случае моделей надежности программного обеспечения «временной интервал» должен выбираться из соображений обеспечения репрезентативности исходных данных. Это означает, что тестирование программ даже в течение длительного промежутка времени с использованием лишь ограниченной области значений входных данных даст несмещенные оценки надежности только в том случае, если в реальных условиях вероятность появления значений входных данных из непроверенной области близка к нулю. Подобное

условие уже рассматривалось при обсуждении требований, предъявляемых к так называемой «программе исследователя» [5].

Модели, предложенные Коркорэном и др. Эти модели заслуживают внимания по следующим причинам. Во-первых, они содержат изменяющиеся вероятности отказов для различных источников ошибок и соответственно разные вероятности их исправления. Во-вторых, они относительно просты с математической точки зрения и допускают простую интерпретацию данных об ошибках в программном обеспечении. В этих моделях не используется такой параметр, как время тестирования, и учитывается только результат N испытаний, в которых наблюдается N_i ошибок i-го типа (относящихся к i-му источнику отказов). Выявленные в ходе N испытаний ошибки i-го типа устраняются с вероятностью a_i. Результатом такого процесса является большее значение показателя надежности, чем доля успешных испытаний, на величину, зависящую от числа обнаруженных ошибок и вероятностей их устранения.

Одна из оценок надежности имеет вид

$$\rho_i = N_0/N + \sum_{i=1}^{K} y_i(N_i - 1)/N \tag{8}$$

где N_0 – число исходов из серии N испытаний, K – априори известное число типов ошибок и

$$y_i = \begin{cases} a_i, \text{если } N_i > 0, \\ 0, \text{если } N_i = 0. \end{cases} \tag{9}$$

В этой модели, как и в других, вероятность a_i должна оцениваться на основании априорной информации или данных предшествующего периода функционирования, и для обеспечения достоверности оценки надежности программного обеспечения в каждом испытании или по крайней мере в серии из N испытаний на вход тестируемой программы должны подаваться данные, соответствующие ее функциональному разрезу [5].

Модель Нельсона. Данная модель была создана в фирме TRW в рамках разработки математической теории надежности программного обеспечения. Эта теория включает точные математические определения основных элементов надежности программного обеспечения, математическое описание связи между

элементами надежности, математические операции над элементами с целью изучения новых аспектов надежности программного обеспечения.

Общее описание модели Нельсона. Интуитивное определение надежности программного обеспечения может быть уточнено в статистическом смысле на основе следующих простых соображений:

- машинная программа p может быть определена как описание некоторой вычисляемой функции F на множестве E всех значений наборов входных данных, таких, что каждый элемент E_i множества E представляет собой набор значений данных, необходимый для выполнения прогона программы: $E = (E_i: i = 1, 2, \dots, N)$;

- выполнение программы p приводит к получению для каждого E_i определенного значения функции $F(E_i)$;

- множество E определяет все возможные вычисления в программе p, т. е. каждому набору входных данных E_i соответствует некоторый прогон p, и наоборот, каждому прогону соответствует некоторый набор входных данных E_i;

- наличие дефектов в программе p приводит к тому, что ей на самом деле соответствует функция F', отличная от заданной функции F;

- для некоторого E_i - отклонение выхода $F'(E_i)$, полученного в результате выполнения программы, от желаемого значения $F(E_i)$ находится в допустимых пределах Δ_i, т. е. $\left|F'(E_i) - F(E_i)\right| \leq \Delta_i$ — для всех остальных E_i, образующих подмножество E_e множества E, выполнение программы p не обеспечивает приемлемого результата, т.е. $\left|F'(E_i) - F(E_i)\right| > \Delta_i$, работа программы прекращается преждевременно или выполнение программы не может закончиться (программа зацикливается). Все такие случаи называются «рабочими отказами».

Каждое E_i представляет возможную комбинацию значений, которые могут быть приписаны входным переменным (переменным, значения которых должны быть зафиксированы в качестве входных данных программы p, для того чтобы сделать возможным ее выполнение). Число N возможных E_i очень велико, но оно конечно, так как с помощью машинного слова фиксированной длины возможно представить лишь конечное число различных значений переменной.

Совокупность действий, включающая ввод E_i, выполнение программы p, которое оканчивается получением результата $F'(E_i)$, или рабочим отказом, называется *прогоном программы* p. Заметим, что значения входных переменных, образующие E_i, не должны все одновременно подаваться на вход программы p. Таким образом, вероятность P того что прогон программы p приведет к рабочему отказу, равна вероятности, что набор входных данных E_i, использованный в данном прогоне, принадлежит множеству E_e. Если обозначить через n_e число различных наборов значений входных данных, содержащихся в E_e, то $P = \frac{n_e}{N}$ есть вероятность того, что прогон программы на наборе входных данных E_i, случайно выбранном из E среди равновероятных, закончится рабочим отказом. При этом $R = 1 - P = 1 - \frac{n_e}{N}$ есть вероятность того, что прогон программы p на наборе входных данных E_i, случайно выбранном из E среди априори равновероятных, приведет к получению приемлемого результата.

Однако в процессе реального функционирования программы выбор входных данных из E обычно осуществляется не с одинаковыми априорными вероятностями, а диктуется определенными условиями работы. Эти условия могут быть охарактеризованы некоторым распределением вероятностей p_i, того, что будет сделан выбор именно входных данных E_i. Множество вероятностей p_i называется функциональным разрезом.

Распределение P может быть определено через p_i с помощью «динамической переменной» y_i, которая принимает значение 0, если прогон программы на наборе E_i заканчивается вычислением приемлемого значения функции, и значение 1, если этот прогон заканчивается рабочим отказом. Тогда $P = \sum_{i=1}^{N} p_i y_i$, есть вероятность того, что прогон программы p на входных данных E_i, выбранных в соответствии с распределением вероятностей p_i, закончится отказом, и $R = 1 - P = \sum_{i=1}^{N} p_i(1 - y_i)$ есть вероятность того, что прогон программы с выходными данными E_i, выбранными в соответствии с распределением вероятностей p_i, приведет к правильному выполнению программы.

Так как R – вероятность того, что единичный прогон программы не закончится отказом на наборе входных данных, выбранных в соответствии с распределением p_i, то вероятность успешного выполнения n прогонов этой программы при независимом для каждого прогона выборе входных данных в соответствии с распределением P будет равна $R(n) = R^n = (1-P)^n$ Таким образом, можно дать следующие математическое определение надежности машинной программы: - это вероятность безотказного выполнения n прогонов программы. Поэтому прогон является единичным испытанием программы.

На практике обычно выбор входных данных для каждого прогона нельзя считать независимым. Исключение составляют лишь такие последовательности прогонов, которые определяются возрастающими значениями некоторой входной переменной или некоторым порядком установленных процедур, как в случае программ, работающих в реальном масштабе времени.

С учетом введенного определения функциональный разрез должен быть переопределен в терминах вероятностей p_{ji} выбора E_i в качестве входных данных при j-м прогоне из некоторой последовательности прогонов. Тогда вероятность того, что j-й прогон закончится отказом, может быть записана в виде $P_i = \sum_{i=1}^{N} p_i y_i$. Надежность R($n$) программы p равна вероятности того, что в последовательности из n прогонов ни один из них не закончится отказом:

$$R(n) = (1-P_1)(1-P_2)\dots(1-P_n) = \prod_{j=1}^{n}\left(1-P_j\right). \qquad (10)$$

Эта формула может быть представлена в виде $R(n) = e^{\sum_{j=1}^{n} 1n(1-P_j)}$ Некоторые из свойств функции R(n) могут стать более зримыми при следующих допущениях:

для $P_j \ll 1$

$R(n) = e^{-\sum_{j=1}^{n} P_j}$

если $P_j = P$ всех j, то $R(n) = e^{-P_n}$

С помощью соответствующих замен переменных и подстановок можно выразить функцию R(n) через время функционирования t. Обозначим через Δt_i

время выполнения j-го прогона программы, через $t_j = \sum_{i=1}^{j} \Delta t_i$ - суммарное время выполнения первых j прогонов программы и причем, что $h(t_j) = -\frac{\ln(1-P_j)}{\Delta t_j}$, тогда $R(n) = e^{-\sum_{j=1}^{n} \Delta t_j h(t_j)}$.

Если величина Δt_j стремится к кулю с ростом n, то сумма в показателе экспоненты становится интегралом и формула для надежности программного обеспечения принимает вид $R(t) = exp\left[-\int_0^t h(s)ds\right]$.

Эта формула хорошо известна в теории надежности технических устройств. В случае функция $h(t_j)$ может быть интерпретирована как функция интенсивности отказов, которая, будучи умноженной на Δt_j, дает условную вероятность проявления отказов в интервале (t_j, $t_i + \Delta t_j$), при отсутствии отказов до момента t_j [5].

Оценка надежности программного обеспечения. Надежность машинной программы может быть оценена путем прогона программы на n наборах входных данных и расчета значения оценки $\hat{R}$ по формуле $\hat{R} = 1 - \frac{\widehat{n_e}}{n}$, где n_e – число наборов входных данных, при которых произошли рабочие отказы.

Если выборка включает n наборов входных данных из E и осуществляется в соответствии с распределением вероятностей p_i, то расчетное значение $\hat{R}$ будет представлять собой несмещенную оценку надежности R в том смысле, что для $p_i \ll 1$, ожидаемое значение R на всем множестве наборов входных данных в пределах выборки равно R. Это может быть показано, если ввести *характеристику выборки* z_{ij}, которая определяется так, что

z_{ij}*=1, если* E_i *входит в выборку* j,

z_{ij}*=0 в противном случае.*

Число различных наборов входных данных n_j, входящих в некоторую выборку j, может быть меньше чем n, так как некоторые наборы E_i, могут быть включены в нее более одного раза. Поэтому $\sum_{j=1}^{N} = z_{ij} = n_j$.

Однако в большинстве случаев число возможных наборов входных данных N настолько больше размера выборки, что повторный выбор каких-либо наборов

маловероятен. Если s_j — вероятность того, что взята выборка j, и M - количество возможных выборок, то $\sum_{j=1}^{M} z_{ij} s_i = 1 - (1 - p_i)^2$.

Так как $\hat{R}$ можно представить в виде $\hat{R_j} = \frac{1}{n}\sum_{i=1}^{N}(1 - y_i)z_{ij}$, что математическое ожидание $\hat{R}$ равно

$$M\left(\hat{R}\right) = \sum_{j=1}^{M} s_j \hat{R}_j = \frac{1}{n}\sum_{i=1}^{M} S_j \sum_{i=1}^{N}(1 - y_i)z_{ij} = \frac{1}{n}\sum_{i=1}^{N}(1 - y_i)\sum_{j=1}^{M} s_j\, z_{ij} =$$

$$\frac{1}{n}\sum_{i=1}^{N}(1 - y_i)[1 - (1 - p_i)^n] = \sum_{i=1}^{N}(1 - y_i)p_i. \qquad (11)$$

При $p_i \ll 1$ $M\left(\hat{R}\right) = 1 - P = R$.

Для получения оценки R должен быть определен функциональный разрез p_i программы. На практике этот разрез определяется путем разбиения всего пространства значений входных переменных на подпространства и нахождения вероятностей того, что выбранный набор входных данных будет принадлежать конкретному подпространству. Определение этих вероятностей основано на оценке вероятностей появления тех или иных входов в реальных условиях функционирования, относительно которых оценивается надежность программы.

Как только вероятности p_i определены указанным образом, случайная выборка из n наборов входных данных, распределенных в соответствии с p_i, может быть получена с помощью некоторого датчика случайных чисел. После реализации n испытаний программы для одних входных наборов данных результаты окажутся правильными, а для других могут быть зафиксированы отказы. При этом процесс испытаний не должен прекращаться, а ошибки не должны исправляться до завершения всех n прогонов; на основании полученных данных может быть вычислена оценка надежности $\widehat{R}$.

Рассмотренный подход к оценке надежности программного обеспечения был проверен на двух программах, написанных различными специалистами на основании одного и того же технического задания. Разрез p_i определялся таким образом, чтобы он приближенно отражал реальные условия функционирования программы. Выборка из 1000 наборов входных данных формировалась в соответствии с этим разрезом с помощью датчика случайных чисел. Было сделано 1000 прогонов каждой программы. Результаты каждого прогона анализировались,

и отмечались те прогоны, которые заканчивались отказом. Для одной программы было зафиксировано три отказа, а для другой - 35. Рассчитанная оценка надежности $\widehat{R}$ для первой программы равнялась 0,997, а для второй - 0,965. Программа, которой соответствовала более высокая надежность, имела более простую структуру, и этот факт подтверждает установившееся мнение, что усложнение структуры программ ведет к снижению их надежности [5].

1.4 Анализ надежностных характеристик программного комплекса встраиваемых систем управления

В теории, надежная эксплуатация автономных беспилотных объектов (АБО) сопровождается корректными выходными данными модулей бортового программного обеспечения (БПО), это значит, что основные функции, выполняемые АБО и связанные, прежде всего, с обеспечением автономной эксплуатации, в том числе решением задач телеметрии и формирования управляющих воздействий, должные иметь оцениваемый и измеримый запас надежности, выраженный рядом характеристик как среды исполнения модулей, так и модельно-алгоритмической компоненты системы управления АБО.

Вопросы расширения и интерпретации классической теории надежности с точки зрения программного обеспечения.

Действия механизмов различных классов неисправностей могут отличаться от физических точек зрения в зависимости от причин их порождающих, при этом одна формулировка может быть использована с точки зрения моделирования надежности и статистической оценки. Одна формулировка имеет ряд преимуществ, как теоретических, так и практических, таких как: более легкое и последовательное моделирование аппаратно-программных систем и аппаратно-программных взаимодействий; адаптивность моделей как для аппаратной надежности, так и для программных систем; математическая согласованность.

Концепция надежности. Рассмотри основные определения, применимые к термину надежность, в части нарушений (ухудшений) надежности, а также средств и атрибутов обеспечения надежности [6].

Надежность определяется как достоверность компьютерной системы, которая позволяет обоснованно полагаться на предоставляемый ею сервис. *Сервис*, предоставляемый системой, является ее поведением, в качестве того как оно воспринимается ее пользователями; *пользователь* - это другая система (человеческая или физическая), взаимодействующая с вышеназванной системой.

В зависимости от приложений, предназначенных для системы, иной акцент может быть положен на различные аспекты надежности, то есть надежность может рассматриваться в соответствии с различными, но взаимодополняющими свойствами, которые позволяют определять *атрибуты* надежности:

- укомплектованность к использованию приводит к *готовности*.

- непрерывность обслуживания приводит к *безотказности*.

- отсутствие катастрофических последствий для окружающей среды ведет к *безопасности*.

- отсутствие несанкционированного разглашения информации ведет к *конфиденциальности*.

- отсутствие ненадлежащего изменения информации ведет к *целостности*.

- способность к ремонту и эволюции ведет к *ремонтопригодности*.

Ассоциирование готовности и целостности в отношении санкционированных действий вместе с конфиденциальностью ведет к обеспечению *безопасности*.

Системный *отказ* происходит, когда предоставляемый сервис отклоняется от выполнения системной функции, последняя из приведенных предназначена для самой системы. *Ошибка* представляет часть состояние системы, которое может привести к последующему сбою: ошибка, влияющая на обслуживание, является признаком того, что происходит или произошел сбой. Неопределенная или гипотетическая причина ошибки является *сбоем (неисправностью)*.

Разработка надежной вычислительной системы требует комбинированного использования набора методов и приемов, которые можно классифицировать на:

- *предотвращение сбоев*: как предотвратить возникновение или внедрение неисправности.

- *устранение сбоев*: как уменьшить присутствие неисправностей (количество, степень).

- *отказоустойчивость*: как обеспечить сервис, способный выполнять функцию системы при наличии неисправностей.

- *прогнозирование сбоев (неисправностей)*: как оценить текущее число, будущую частоту и последствия неисправностей.

Введенные понятия могут быть сгруппированы в три класса:

- *Нарушения надежности*: сбои, ошибки, отказы; они нежелательны - но, в принципе, они не являются неожиданными обстоятельствами, вызывающими или приводящими к ненадежности (определение которых очень просто вытекает из определения надежности: уверенность не может быть или не будет длительно полагаться на сервис).

- *Средства* обеспечения надежности: предотвращение сбоев (неисправностей), устранение неисправностей, отказоустойчивость, прогнозирование неисправностей; это методы и приемы, позволяющие одному предоставить возможность предоставления сервиса, на которую можно полагаться, и достичь уверенности в этой способности.

- *Атрибуты* надежности: готовность, безотказность, безопасность, конфиденциальность, целостность, ремонтопригодность; эти атрибуты позволяют выразить свойства, ожидаемые от системы, и позволяют оценить качество системы, являющееся следствием применяемых в ней средств, которые противостоят ухудшениям.

О нарушении надежности. Первостепенное значение имеют нарушения надежности, поскольку мы должны знать, с чем мы сталкиваемся. Механизмы создания и проявления сбоев, ошибок и отказов могут быть обобщены следующим образом:

1. Сбой (неисправность) активна, когда она вызывает ошибку. Активный сбой - это либо внутренняя неисправность, ранее спящая и активируемая

процессом вычисления, либо внешняя неисправность. Большинство внутренних неисправностей происходят между их спящим и активным состояниями. Физические неисправности могут напрямую влиять только на аппаратные компоненты, тогда как антропогенные неисправности могут влиять на любой компонент.

2. Ошибка может быть скрытой или обнаруженной. Ошибка скрыта, если она не была распознана как таковая; ошибка обнаруживается алгоритмом или механизмом обнаружения. Перед обнаружением ошибка может исчезнуть. Ошибка может, и в целом, распространяться; при распространении ошибка создает другие новые ошибки. Во время работы наличие активных сбоев определяется только обнаружением ошибок.

3. Отказ возникает, когда ошибка проходит через интерфейс система-пользователь и влияет на обслуживание, предоставляемое системой. Отказ компонента приводит к неисправности для системы, которая содержит компонент, и при просмотре другим компонентом (компонентами), с которым он взаимодействует; режимы отказа вышедшего из строя компонента затем становятся типами сбоев для компонентов, взаимодействующих с ним.

Некоторые примеры, иллюстрирующие патологию сбоя:

- Результатом ошибки программиста является (спящий) сбой в написанном программном обеспечении (неисправная команда или данные); при активации (вызов компонента, в котором находится неисправность, и инициирование неисправной команды, последовательности команд или данных с помощью соответствующего шаблона ввода), неисправность становится активной и выдает ошибку; если и когда ошибочные данные влияют на предоставляемый сервис, происходит отказ.

- Короткое замыкание, происходящее в интегральной схеме, является отказом. Следствие (соединение, удерживаемое в логическом значении, модификация функции цепи и т.д.) является неисправностью, которая будет оставаться бездействующей, пока она не была активирована, продолжение

процесса будет идентично продолжению процесса, аналогичному предыдущему примеру.

- Неправильное человеко-машинное взаимодействие, выполняемое оператором во время работы системы, является неисправностью (с точки зрения системы); полученные измененные обработанные данные являются ошибкой.

- Ошибки в руководстве пользователя по обслуживанию или эксплуатации могут привести к неисправности в соответствующем руководстве (ошибочные директивы), которое останется бездействующим до тех пор, пока директивы не будут приняты во внимание, чтобы иметь дело с конкретной ситуацией.

Основные характеристики надежности суммируются в виде дерева, представленного на рисунке 4 [6].

Рисунок 4 – Дерево надежности

На рисунке 5 приведена классификация сбоев (неисправностей); верхняя часть указывает точку зрения, в соответствии с которой они классифицируются, а нижняя часть дает вероятные комбинации в соответствии с этими точками зрения, а также обычную маркировку этих комбинаций, а не их определение.

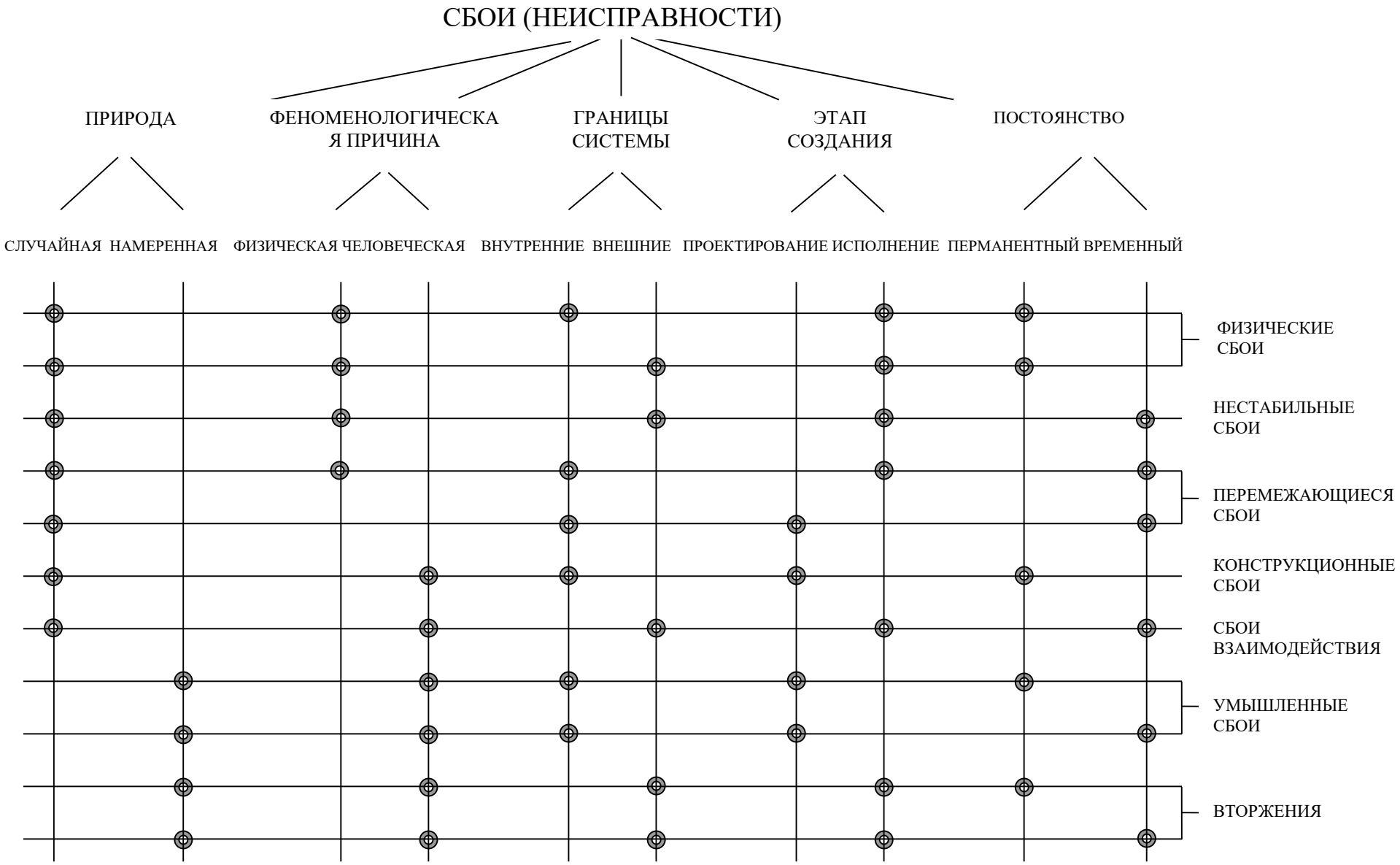


Рисунок 5 – Классификация сбоев (неисправностей)

Примечательно, что даже само понятие сбоя является случайным, а в действительности средством, обеспечивающим остановку рекурсии, вызванной причинно-следственной связью между сбоями, ошибками и отказами - отсюда и определение: признанная или гипотетическая причина ошибки. Эта причина может варьироваться в зависимости от выбранной точки зрения: механизмов отказоустойчивости, инженеров по обслуживанию, сервисной службы, разработчика, физика полупроводников и т.д. Фактически, сбой - это не что иное, как следствие отказа какой-либо другой системы (в том числе разработчика), которая предоставила или в настоящее время предоставляет сервис данной системе.

Система не может, и, как правило, не должна, всегда терпеть неудачу путем возникновения одних и тех же неисправностей. Пути, приводящие к выходу из строя системы, могут быть вызваны её отказами. Их можно охарактеризовать по трем направлениям, как показано на рисунке 6.

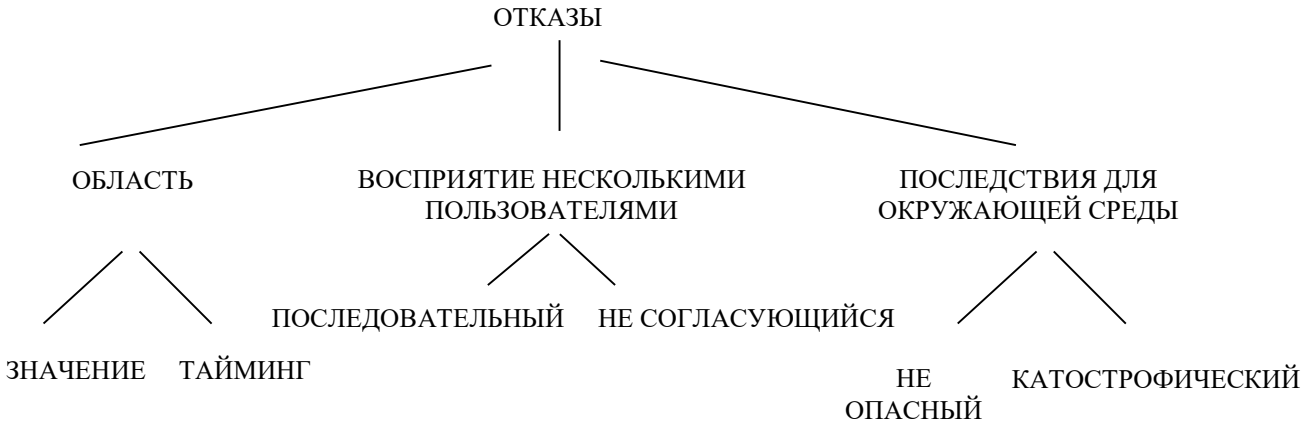


Рисунок 6 – Выявленные направления отказов

Об атрибутах надежности. Определение, данное для термина целостность - предотвращение неправомерного изменения информации, обобщает обычные определения (например, предотвращение несанкционированного изменения или удаления информации, или обеспечение одобренного изменения данных), которые непосредственно связаны с определенным классом сбоев, то есть преднамеренных сбоев

(намеренно недоброжелательных действий). Данное определение также включает в себя случайные сбои (то есть, сбои которые появляются или создаются случайно), и использование слова информации предназначено, чтобы не ограничиваться строго данными: целостность программ также является существенной проблемой; в отношении случайных сбоев, восстановление ошибок действительно направлено на восстановление целостности системы.

Конфиденциальность, а не безопасность, была введена как основной атрибут надежности. Обычно безопасность определяется как сочетание конфиденциальности, целостности и готовности, когда эти три понятия воспринимаются в отношении несанкционированных действий. Определение безопасности, включающее три аспекта, состоит в: предотвращении несанкционированного доступа и / или обработки информации; в вопросах безопасности действительно доминируют преднамеренные сбои, но не ограничиваются ими: случайный (например, физический) сбой может вызвать непредвиденную утечку информации.

Дано определение ремонтопригодности - способность проходить ремонт и эволюцию, сознательно выходит за рамки корректирующего обслуживания, что относится только к ремонтопригодности. Способность эволюционировать очевидно относится к двум другим формам обслуживания, то есть: адаптивное обслуживание, которое приспосабливает систему к изменениям окружающей среды и совершенное обслуживание, что улучшает функции системы, реагируя на изменения, определенные заказчиком и проектировщиком, определяющим изменения. Однако граница между ремонтопригодностью и способностью к развитию не всегда ясна (например, если запрошенное изменение направлено на исправление сбоя спецификации). Ремонтопригодность фактически обеспечивает надежность при рассмотрении всего срока эксплуатации системы: системы, которые не подвергаются

адаптивному или совершенному обслуживанию, скорее всего, будут исключениями.

Свойства, позволяющие определять атрибуты надежности, могут быть сформулированы в большей или меньшей степени в зависимости от приложения, предназначенного для соответствующей компьютерной системы. Например, всегда востребована готовность (хотя и в различной степени в зависимости от приложения), тогда как надежность, безопасность и конфиденциальность могут быть востребованы или не востребованы в соответствии с приложением. Требования к атрибутам надежности оказывают прямое влияние на соответствующий баланс средств, которые будут использоваться для получения надежной системы. Эту проблему еще труднее устранить, поскольку определенные атрибуты являются антагонистическими (например, готовность и безопасность, готовность и конфиденциальность) и требуют компромиссных решений. Учитывая три основных расчетных параметра компьютерной системы (например, стоимость, производительность и надежность), проблема еще более усугубляется тем, что процесс измерения надежности не так хорошо освоен, как проектирования с точки зрения затрат.

О средствах обеспечения надежности. Все практические советы, которые появляются в основных определениях, приведенных выше, на самом деле цели, которые не могут быть полностью достигнуты, поскольку все соответствующие виды деятельности осуществляются людьми и поэтому подвержены несовершенствам. Эти недостатки приводят к зависимостям, объясняющим, почему только комбинированное использование вышеперечисленных методов - предпочтительно на каждом этапе процесса проектирования и реализации, что может наилучшим образом привести к надежной вычислительной системе. Эти зависимости можно охарактеризовать следующим образом: несмотря на предотвращение сбоев с помощью методологий проектирования и правил построения (несовершенных, чтобы

быть работоспособными), создаются сбои - отсюда необходимость устранения сбоев.

Само по себе устранение сбоев является несовершенным, как и все готовые системные компоненты - аппаратное или программное обеспечение, отсюда и важность прогнозирования сбоев. Наша растущая зависимость от вычислительных систем влечет за собой необходимость отказоустойчивости, которая, в свою очередь, основана на правилах построения - следовательно, на устранении сбоев, прогнозировании сбоев и т.д.

Следует отметить, что этот процесс еще более рекурсивный, чем кажется из вышесказанного: современные компьютерные системы настолько сложны, что их проектирование и реализация требуют компьютеризованных инструментов, чтобы быть экономически выгодными (в широком смысле, включая возможность приемлемых временных затрат). В свою очередь, сами эти инструменты должны быть надежными.

1.5 Исследование методов повышения надежности программного обеспечения

Процессы разработки и методы проектирования программного обеспечения изучаются в течение многих десятилетий. Несмотря на это, до сих пор нет надежных средств, гарантирующих безотказную работу сложных программных систем. В действительности, никогда не случится так, что мы будем в состоянии гарантировать безошибочную работу программного обеспечения. Причина заключается в том, что два основных способа продемонстрировать отлаженную работу программного обеспечения - это доказательство корректности программ и исчерпывающего тестирования, которые никогда не могут быть реализованы на практике в системах, базируемых на крупном комплексном программном обеспечении. С другой стороны, пересмотр данных подходов в отношении надежного программного

обеспечения блочного (объектного) построения может направить нас по верному пути к достижению поставленной цели.

Подходы к доказательству отлаженной работы программного обеспечения имеют тенденцию в отношении работы относительно небольших и простых синхронных систем, в то время как методы тестирования, все более и более нестандартные, не гарантируют получение безошибочного кода, поскольку методы исчерпывающего тестирования не является реализуемыми в большинстве случаев. Таким образом, целесообразно исследовать методы, которые позволяют программным системам функционировать надежно и безопасно, даже если присутствуют потенциальные отказы.

Подходом к обработке неисправностей, априорно неизвестного и непредсказуемого, как аппаратного, так и программного обеспечения является отказоустойчивость. За последние три десятилетия наблюдается значительное количество исследований, а также практических инженерных решений - программного обеспечения, в этой области. Рассмотрим некоторые элементарные подходы, которые лежат в основе проектирования и реализации отказоустойчивого программного обеспечения, основанного на принципе разнообразия программного обеспечения, то есть, обеспечении отказоустойчивости за счет функциональной избыточности.

1.5.1 Современное состояние

Отказоустойчивое программное обеспечение было предложено для использования в ряде важнейших областей применения. Например, на атомных электростанциях [7], в железнодорожных системах [8], а также в аэрокосмических системах [9, 10, 11]. Обзорные части использования разнообразия программного обеспечения в автоматизированных системах управления можно найти в работах [6,12].

Ряд систематических экспериментальных исследований проблем отказоустойчивого программного обеспечения был проведен в на протяжении

последних 30 лет, параллельно, как научными учреждениями, так и промышленностью. Например, проводились эксперименты, связанные с использованием отказоустойчивого программного обеспечения для применения в ядерной промышленности [13], аэрокосмической промышленности [14, 15, 16], а также в других областях [17].

Считается, что первое промышленное использование отказоустойчивого программного обеспечения на основе принципа разнообразия, произошло в железнодорожной системе [18]. Ряд организаций использовали данный подход в целях содействия развитию, проверке, или для реализации оперативных железнодорожных приложений для развертывания в Швеции, Дании, Финляндии, Швейцарии, Турции и Болгарии, Италии [19], Сингапуре [20], и Соединенных Штатах Америки [21].

Использованию отказоустойчивого программного обеспечения в авиационно-космической промышленности было уделено существенное внимание на протяжении многих лет. Данный подход был рассмотрен и реализован как в военных (например, [22]), так и гражданских (например, [23]) воздушных судах и в космических кораблях [24]. Например, система управления закрылками для гражданского авиалайнера Airbus A310 состоит из двух функционально идентичных компьютеров с разнообразным аппаратным и программным обеспечением [25] Кроме того, отказоустойчивость является неотъемлемой частью одной из экспериментальных технологий полета, так называемой технологии «с обратной стреловидностью крыла», недостатком которой является аэродинамическая неустойчивость, но при этом как достоинство - высокая маневренность, [26].

Другим примером является космический челнок NASA. Шаттл имел конфигурацию четырех идентичных полетных компьютеров, в каждый из которых было загружено идентичное программное обеспечение для борьбы со сбоями оборудования, а также пятый компьютер, разработанный сторонним производителем, программное обеспечение которого отличалось от других, но

при этом частично сохраняло функциональную эквивалентность. Исполнение программного обеспечения данного компьютера начиналось в случае, если программное обеспечение в остальных четырех компьютеров не могло достичь консенсуса в критических фазах полета [6,27]. Принцип разнообразия программного обеспечения стал важным аспектом во всех вопросах, упомянутых выше.

Практические опыты с отказоустойчивым программным обеспечением представляются весьма неоднозначными, при этом, они более положительные, чем отрицательные, хотя некоторые вопросы остаются нерешенными. Общий консенсус в том, что отказоустойчивое программное обеспечение имеет потенциал для повышения надежности компьютерной системы. К открытым и спорным вопросам относятся следующие, как и на сколько отказоустойчивое программное обеспечение повышает надежность системы на практике [28], следует ли отказоустойчивое программное обеспечение использовать на всех критически важных системах [29], и что использует механизм отказоустойчивости в ресурсном плане и насколько рентабельно это [30].

Основная причина этих сомнений имеет отношение к методам разработки программного обеспечения, основанным на избыточности, что является потенциальной возможностью возникновения неисправностей общего вида и коррелируемых в совпадении отказов среди элементов программного обеспечения, которые обеспечивают избыточность.

В отличие от аппаратных сбоев, неисправности программного обеспечения являются по большей части результатом спецификации программного обеспечения и ошибок проектирования, и, таким образом, простая репликации программных компонентов не обеспечивает достаточной защиты. Данное обстоятельство диктует потребность в стремлении к проектированию и разработке методов, которые поощряют использование различных алгоритмов в избыточных компонентах и минимизируют потенциальную возможность возникновения совпадающих неисправностей.

Многие эксперименты с отказоустойчивым программным обеспечением фиксировали коррелируемые отказы среди версий программного обеспечения, обеспечивающих избыточность (например, [6,31,32]. Данное обстоятельство вызывает некоторое предположение, что корреляционная зависимость отказов является устойчивой функцией методов и процессов разработки программного обеспечения, используемых в разработке программной версии [33]. Совершенствование процесса разработки программных версий может эффективно повысить общую надежность системы.

Основным направлением введения отказоустойчивого программного обеспечения в приложение является обеспечение метода динамического определения, связанного с корректным выходным результатом программного обеспечения, то есть функции самопроверки [34]. Это часто достигается за счет сочетания различных, но функционально эквивалентных программных альтернатив, версий или вариантов и сравнения времени выполнения версий до получения результатов. Однако, другие технологии, начиная от проверки математической согласованности до кодирования, также применимы [35], как и методы, которые используют принцип разнообразия данных [36].

Когда программное исполнение сталкивается с программными ошибками или дефектами, очень часто система делает переход в ошибочное (внутреннее) состояние, то есть, будет создан внутренний результат, отличающийся от ожидаемого. Если этот ошибочный результат распространяется и в конечном счете зафиксируется пользователем системы, то в данном случае имеет место сбой системы [37]. После того, как ошибочное состояние было определено, может быть начато восстановление после ошибок. Данная процедура может включать в себя «обратное восстановление», то есть состояние системы сохраняется в заранее определенных контрольных точках восстановления, при этом в случае обнаружения ошибочного состояния, систему возможно восстановить из

более ранней, сохраненной точки восстановления, а затем перезапустить из этого состояния.

Альтернативный подход заключается в прямом восстановлении. Прямое восстановление может быть реализовано как переход в новое состояние системы, в котором программное обеспечение может работать (часто в ограниченном режиме), или путем компенсации ошибок, основанном на алгоритме, который использует избыточность, встроенную в систему, с целью извлечения корректного ответа. Особым случаем последнего является постоянное маскирование ошибок, когда компенсация имеет место независимо от того, обнаружена ошибка или нет (например, некоторые формы голосования) [30]. Существуют комбинации прямого и обратного восстановления также имеет место быть.

Важным фактором в определении, выявлении и обработке ошибки является знание о том, будут ли результаты ошибочных состояний алгоритмическими или использующими память или нет. Алгоритм или программа называется "без памяти", если она использует только полученные или произведенные текущие данные. Примером алгоритма «без памяти» является мониторинг технологического процесса, где задачи начинаются на основе текущих данных датчиков, и не используют данные из предыдущей обработки.

Важным критерием для оценки приемлемости схем реализации отказоустойчивости, является обработка накладных расходов, требуемых для реализации отказоустойчивости, содержательности механизма обнаружения ошибок, определяется ли правильность результата в абсолютной или относительной форме (например, в соответствии со спецификацией, или другой версией), является ли схема реализации последовательной или параллельный, включает ли контроль ошибок приостановление обслуживания или нет, являются ли результаты представленными в течение ограниченного

времени, и при таком количеством ошибок и какой ценой можно обеспечить устойчивость.

1.5.2 Моноверсионные модели повышения надежности

Приемочный тест. Одним из основных подходов к самопроверке является внутреннее приемочное испытание. Приемочный тест реализуется программистом в соответствии с предоставляемыми спецификацией и механизмом обнаружения ошибок, что обеспечивает проверку результатов выполнения программы. Приемочный тест может содержать простые тесты и ограничения с целью определения заявленного уровня приемлемости (корректности). К примеру, гораздо проще разрабатывать программное обеспечение, которое определяет что «список» отсортирован, чем разрабатывать программное обеспечение, которое выполняет функцию сортировки. Тем не менее, в целом приемочный тест может быть комплексным и дорогостоящим, но необходимым для реализации полного решения проблемы. Важной характеристикой приемочных тестов является то, что они используют только те данные, которые также доступны в программе во время выполнения (текущие данные).

Интересным примером является следующая неявная спецификация функции квадратного корня [38], *SQRT*, который может служить в качестве приемочного теста:

$$\text{for } (0 \le x \le y)\ (\text{ABS}((\text{SQRT}(x)*\text{SQRT}(x))-x)<E) \qquad (12)$$

где E допустимый диапазон ошибки, а x и y действительные числа. Если E вычисляется для фактического кода, сформированного для программы *SQRT*, и технические характеристики машины известны программе, то уравнение (12) может служить в качестве полного теста самоконтроля (приемочного теста). Если E известно только программисту (или тестеру), а также во время выполнения программы нет доступа к техническим характеристикам аппарата или информации о допустимом распространении

ошибки в пределах программы, то тест либо не полный, либо не может быть выполнен самой программой за отведенное на это время.

Приемочные тесты, которые специально созданы для алгоритма иногда называют алгоритмическими [39]. Например, предоставление контрольных сумм для строк и столбцов матрицы может облегчить обнаружение проблем и последующее восстановление [40]. Аналогичным образом, использование избыточных звеньев в связанных списках могут способствовать борьбе с частичной потерей элементов списка [6,41].

Внешняя согласованность (целостность).

Внешняя проверка целостности представляет собой расширенный механизм обнаружения ошибок. Он может быть использован, с целью принятия решения о правильности результатов выполнения программы, но только с некоторым внешним вмешательством. Данный подход способствует развитию методов тестирования программного обеспечения, а также может быть использован для обработки исключений во время выполнения программных компонентов. В случаях, когда возникают трудности заблаговременного вычисления верного ответа, данный подход может быть наиболее экономически эффективным способом проверки компонентов отказоустойчивой системы, или проверки на результат во время выполнения программных компонентов.

Проверка согласованности может использовать всю информацию, включая информацию, которая может быть недоступна для программы во время ее исполнения, но которая может быть доступна внешнему агенту. Примерами являются сторожевые процессы, которые контролируют выполнение программных систем и используют информацию, которая может быть недоступна программному обеспечению, таких как тайминг (синхронизация), для обнаружения и устранения проблем [42]. Другим примером является периодический ввод (ручной или автоматический) координат местоположения в навигационную систему. Еще одним примером

является сигнал об исключении, вызванный компьютерным оборудованием или операционной системой при обнаружении переполнения стека, нехватки разрядности при работе с плавающей запятой или целых чисел, деления на ноль.

Сигнал прерывания или исключение, которые представляют собой аномальные события, происходящие в компьютерной системе, производят представление как об особо полезном и распространенном ресурсе для проверки внешней согласованности. Часто эти сигналы могут быть обнаружены и захвачены программным обеспечением, и исключения могут быть обработаны для обеспечения формы отказоустойчивости.

Например, многие современные языки программирования допускают захват исключений с плавающей запятой, исключения по результату деления на ноль или арифметические сигналы IEEE (например, NANs1) [43] и вызов соответствующих обработчиков исключений, которые предоставляют альтернативные вычисления или другие действия, и таким образом защищают пользователей от этого типа ошибки во время выполнения.

Проверка целостности может включать сравнение с точными результатами, но чаще данное обстоятельство предполагает использование знаний о точной природе входных данных и условий в сочетании со знанием преобразования (отношения) между входными данными и выходными данными. Соотношение согласованности должно быть достаточным для подтверждения корректности.

Например, предположим, что навигационное программное обеспечение самолета считывает показания акселерометра, из которых производятся вычисления, что является оценкой ускорения воздушного судна [44]. Пусть оценка вектора ускорения $\hat{x}$ задается методом наименьших квадратов.

$$\hat{x} = [C^T C]^{-1} C^T y \qquad (13)$$

Матрица C представляет собой матрицу преобразования из структуры объекта в навигационную систему отсчета, C^T - ее транспонированную, -1

обозначает обратную матрицу, а измерения датчика связаны с истинным вектором ускорения *x* по формуле

$$y = Cx+\tilde{y} \quad (14)$$

где $\tilde{y}$ - погрешность датчика, вызванная шумом, смещением и квантованием. Тогда

$$C^T C(\hat{x} - x) = C^T \tilde{y} \quad (15)$$

является критерием, способным утверждать корректность для оценки ускорения. При этом, x и $\tilde{y}$ обычно недоступны для навигационного программного обеспечения.

Однако, если мы предоставим всю информацию, в том числе, относящуюся к окружающей среде, мы можем управлять *x* и $\tilde{y}$ и обнаруживать проблемы с алгоритмами без предварительного знания корректного ответа. Это может обеспечить ряд возможностей во время тестирования в автономном режиме, когда окружающая среда полностью находится под контролем. Разумеется, такой тест не может быть применен во время эксплуатационного использования программного обеспечения, пока точный статус окружающей среды не будет предоставлен независимо.

Напротив, исключения аппаратного обеспечения и операционной системы, такие как переполнение и опустошение буфера, могут и должны обрабатываться во время выполнения программных компонентов, и соответствующие алгоритмы обработки исключений должны быть частью любой программной системы, которая стремится обеспечить степень отказоустойчивости.

Автоматическая проверка численных результатов

Весьма специфичным набором методов для динамического обнаружения и контроля численных отказов является автоматическая проверка численной точности. Этот тип проверки обрабатывает ошибки, результирующие из алгоритмической микропамяти, то есть распространение ошибок в пределах численного алгоритма и, возможно, обрабатывает

нестабильность численного алгоритма. По существу, проблема состоит в том, что проверка численного алгоритма ни в коем случае не гарантирует его численную корректность, если его численные свойства не будут явно верифицированы.

Арифметика с плавающей точкой - очень быстрый и довольно часто используемый подход к научным и инженерным расчетам. Как правило, отдельные операции с плавающей запятой, доступные в современных компьютерах, являются максимально точными, и все же вполне возможно, что надежность численного программного обеспечения будет достаточно низкой, потому что ряд последовательных операций с плавающей запятой дает совершенно неправильные результаты из-за ошибок округления.

Чтобы проиллюстрировать проблему, рассмотрим следующий пример. Пусть *x* и *y* - два вектора с шестью компонентами, x = (10^{20}, 12^{23}, 10^{24}, 10^{18}, 3, -10^{21}) и y = (10^{30}, 2, -10^{26}, 10^{22}, 2111, 10^{19}). Скалярный результат этих двух векторов определяется как

$$x * y = \sum_{i=1}^{6} x_i * y_i \qquad (16)$$

Правильный ответ – 8779. Однако реализация этого выражения практически на каждой доступной сегодня платформе будет возвращать ноль, если не будут приняты специальные меры предосторожности. Причиной является округление, связанное с большой разницей в порядке величины слагаемых. Это происходит несмотря на то, что каждое индивидуальное число может быть достаточно удобно представлено в формате с плавающей точкой для всех платформ.

Можно построить более простые и более сложные примеры, показывающие, что простые численные алгоритмы, такие как метод Ньютона, могут стать очень неустойчивыми, если обычная арифметика с плавающей запятой используется без явного контроля распространения ошибок. Проблема распространения арифметических ошибок может стать очень острой в таких приложениях, как моделирование и математическое

моделирование, и усугубляется современными высокоскоростными компьютерами.

Решение, предложенное в [45, 35], основано на вычислении оптимального точечного произведения с использованием накопления с фиксированной точкой для обеспечения максимальной вычислительной точности на интервальной арифметике (что подразумевает определенные правила округления) для точного вычисления верхней и нижней границы результата и методы автоматической дифференциации.

Если вычисленный результат не может быть проверен на предмет корректности (например, вычисленные границы слишком велики), пользователю может быть предоставлена опция предоставления альтернатив алгоритмам и методам или возможность перехода к высокоточной арифметике, за которой следуют повторное подтверждение результатов и решение о их приемлемости. Этот подход динамически выявляет и контролирует данный тип сбоя. Доступны также другие решения в вопросе проектирования критичных систем, использующих численное программное обеспечение (например, [46]).

1.5.3 Программная избыточность

Методика использования избыточных программных модулей в качестве защиты от остаточных ошибок программного обеспечения была унаследована от аппаратного обеспечения. Аппаратные ошибки обычно случайны (например, из-за старения компонента). Таким образом, использование идентичных блоков резервного копирования или резервных запасных устройств с автоматической заменой отказавших компонентов во время выполнения является разумным подходом (например, 47,48). Однако репликация версии программного обеспечения для обеспечения избыточности резервного копирования имеет ограниченную эффективность, поскольку

ошибки программного обеспечения практически всегда связаны с проектированием и реализацией, и поэтому также будут тиражироваться.

Результирующим эффектом будет то, что порождение реплицированного сбоя приведет к одновременному отказу всех версий что не позволит обеспечить отказоустойчивость. Кроме того, существуют временные или переходные ошибки, которые часто возникают из-за сложного взаимодействия между оборудованием, прикладным программным обеспечением и операционной системой. Такие сбои (называемые Гейзенбаг в [6,49]) редко могут быть дублированы или диагностированы. Гейзенбаг (англ. heisenbug) — термин, используемый в программировании для описания программной ошибки, которая исчезает или меняет свои свойства при попытке её обнаружения. Обычным решением является повторное выполнение программного обеспечения в надежде, что переходное возмущение закончится.

Решение, предлагаемое специально для программного обеспечения, заключалось в том, чтобы независимые производители выпускали функционально эквивалентные программные компоненты [50,51]. Предполагалось, что разные производители будут использовать разные алгоритмы и детали реализации, а остаточные ошибки одного независимого производителя программного обеспечения будут отличаться от остаточных ошибок другого производителя; поэтому при сбое одной из версий его резервная или запасная копии не будет выходить из строя при тех же входных данных и условиях и станет основным модулем или, по крайней мере, может форсировать сигнал разногласия. Цель состоит в том, чтобы сделать модули максимально разнообразными насколько это возможно. Общая философия заключается в том, чтобы повысить вероятность отказа модулей на непересекающихся подмножествах входного пространства и, таким образом, иметь в любой момент, по крайней мере, один правильно функционирующий программный компонент.

Спецификации, используемые в этом процессе, сами могут быть разнообразными до тех пор, пока сохраняется окончательная функциональная эквивалентность продуктов. Спецификация указывает на определенные критические выходные данные, которые должны быть представлены программе принятия решения, чтобы определить, работает ли система правильно. Каждый разработчик создает программный модуль или версию, которая реализует спецификацию и обеспечивает выходы, указанные спецификацией. Избыточность требует возможности судить о приемлемости результатов нескольких модулей либо путем прямой оценки, либо путем сравнения.

Алгоритм, который сравнивает или оценивает выходы, называется арбитром. Такая программа может быть очень простой или очень сложной в зависимости от приложения и поэтому также может быть источником ошибок. Программа арбитража может использовать выходные данные из избыточных версий, чтобы определить, какие из них (если таковые имеются) являются правильными или безопасными для перехода к следующей фазе программного обеспечения. Решение может быть основано на нескольких различных алгоритмах. Предлагается несколько методов принятия решения. Это включает в себя голосование, выбор медианного значения и приемочное тестирование, а также более сложное принятие решений. Во всех ситуациях, конечно, возникают проблемы, если отказы являются совпадающими, коррелированными или подобными.

1.5.4 Отказы и сбои

Мы используем термины совпадающие, коррелированные, зависимые или межверсионные отказы в следующих случаях. Когда два или более функционально эквивалентных компонента программного обеспечения выходят из строя в одном и том же случае ввода, мы говорим, что произошел совпадающий сбой. Отказ от k компонентов увеличивает k-кратный

совпадающий сбой. Когда две или более версий дают один и тот же неправильный ответ (с заданным допуском), мы говорим, что был получен идентичный и неправильный ответ (ИИН).

Если измеренная вероятность совпадающих сбоев существенно отличается от ожидаемой случайной вероятности, обычно основанной на измеренных вероятностях отказа участвующих компонентов, то мы говорим, что наблюдаемые совпадающие отказы коррелированны. Два события могут коррелироваться, потому что они напрямую зависят друг от друга или потому, что оба они зависят от некоторых других, но одинаковых событий (косвенных зависимостей), или того и другого.

Пусть $P\{\}$ - вероятность. Тогда

$$P\{\text{сбой версии}(i) \mid \text{сбой версия}(j)\} \neq P\{\text{сбой версии}(i)\} \quad (17)$$

Означает, что условная вероятность того, что версия *i* дала сбой, если версия *j* дает сбой, отличается от вероятности того, что версия *i*, рассмотренная сама по себе, дает сбой на одних и тех же входах. Если это соотношение истинно, у нас нет независимости сбоев [52]. Можно сказать, что несколько компонентов содержат одну и ту же или подобную ошибку или общую неисправность, если природа неисправности, а также переменные и функции, которые она затрагивает, одинаковы для всех задействованных компонентов. Результат выполнения при общих причинах сбоев может быть идентичным (ИИН с точностью до допуска) или может отличаться. Также возможно, что различные ошибки приводят к случайному совпадающему сбою, давая либо разные ответы, либо ответы ИИН.

Возможные события отказа показаны на рисунке 7. Диаграмма Венна, показанная на рисунке, представляет собой перекрытие в пространстве отказов трех функционально эквивалентных программных единиц (или 3-х кортежей вариантов). Незаштрихованные области - это области, в которых один из трех компонентов (программ P1, P2, P3) выходит из строя при данном вводе. В слегка заштрихованных областях показаны области, в которых два из

трех компонентов дают случайный сбой, а темная заштрихованная область посередине является областью, где все три компонента дают случайный сбой. Особый интерес представляют области, отмеченные черным цветом, которые представляют события, когда компоненты вырабатывают ИИН ответы.

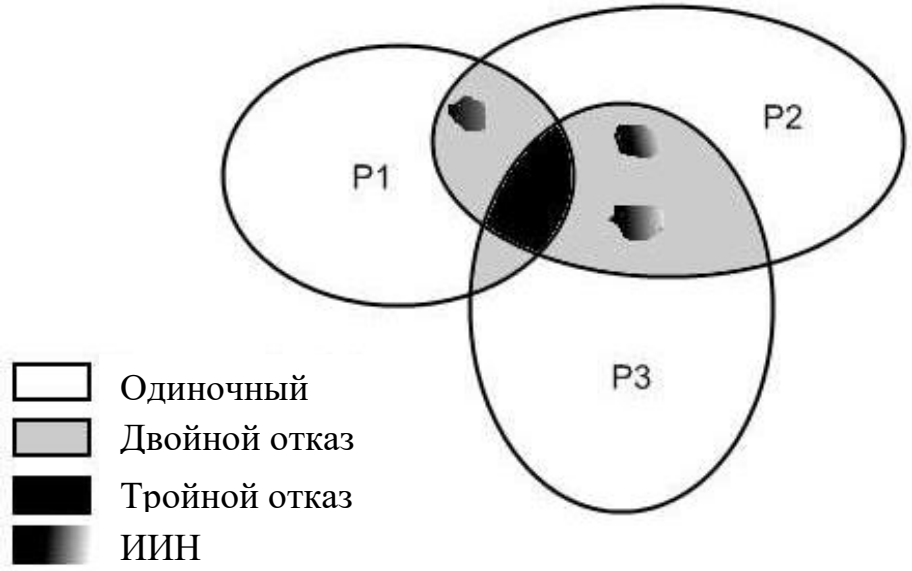


Рисунок 7 - Иллюстрация пространства отказов для трех функционально эквивалентных программ

1.5.5 Базовые методы мультиверсионного подхода

Наиболее распространены две отказоустойчивые программные схемы, это N-версионное программирование [6,53] и модель восстанавливающихся блоков [54]. Обе схемы основаны на избыточности программных компонентов и допущение, о том, что совпадающие сбои компонентов редки, а когда они происходят, ответы являются достаточно разнородными, поскольку механизм принятия решения о правильности ответа не является двусмысленным.

Отказоустойчивые программные механизмы, основанные на избыточности, особенно хорошо подходят для параллельных сред обработки, где параллельное выполнение избыточных компонентов может значительно уменьшить, иногда непомерно высокие затраты, связанные с их последовательным выполнением [54]. Также доступны некоторые гибридные методы, такие как согласованные восстанавливающиеся блоки [55],

контрольные точки [51], совместное восстановление после ошибок [56], варианты N-версионного программирования [57] и некоторые частичные отказоустойчивые подходы [58].

Восстанавливающиеся блоки

Одной из первых отказоустойчивых схем программного обеспечения, использующих мультиверсионный программный подход, являются восстанавливающиеся блоки (ВБ) [59]. Модуль принятия решения является приемочным тестом (ПТ). Процесс начинается, когда выход первого модуля проверяется на приемлемость. Если приемочный тест определяет, что выход первого модуля неприемлем, то он восстанавливает или «откатывает» состояние системы до выполнения первого или основного модуля. Затем модуль принятия решения позволяет второму модулю выполнить свою функцию и оценивает его выход и т.д. Если все модули выполняются и ни один из них не выдаёт приемлемых выходов, система возвращает ошибку. Рисунок 14 иллюстрирует технику.

Одной из проблем этой стратегии в однопроцессорной среде является последовательный характер выполнения версий [30], хотя в распределенной среде модули и приемочные испытания могут выполняться параллельно. Блок распределенного восстановления подробно обсуждается в [52]. Другая потенциальная проблема - найти простой и высоконадежный приемочный тест, который не предполагает разработку дополнительной версии программного обеспечения.

Форма приемочного теста зависит от приложения. Как показано на рисунке 8, возможны разные приемочные тесты для каждого модуля, но на практике обычно используется только один.

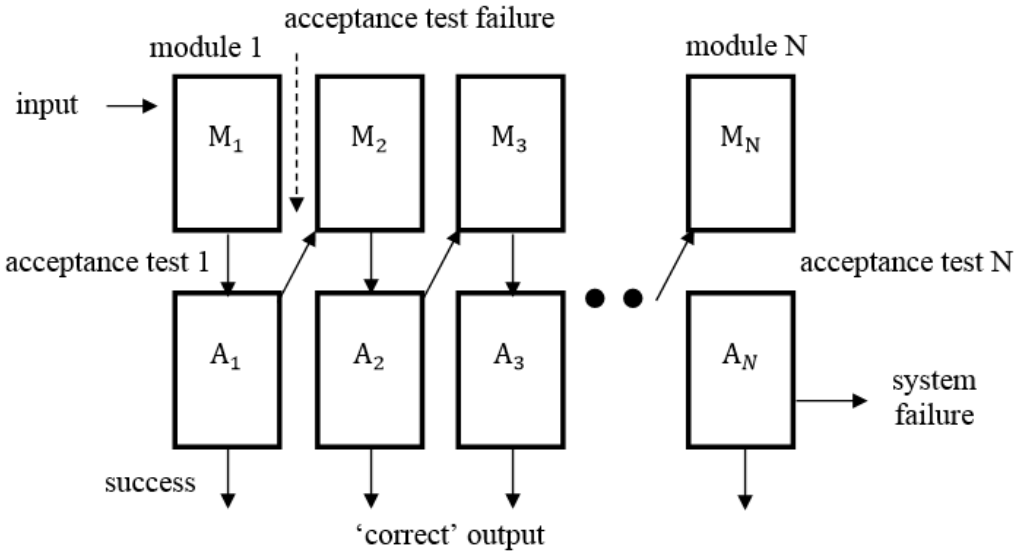


Рисунок 8 – Восстанавливающиеся блоки

Крайним случаем приемочного теста является еще один полноценный модуль, и приемочный тест будет состоять из сравнения выходов проверяемого модуля с вычисленным приемочным тестом. Это было бы эквивалентно поэтапной схеме программирования с двумя версиями, где один из выходов на каждом этапе всегда был бы из одной и той же версии (приемочный тест).

N-версионное программирование

N-версионное программирование [51] предлагает параллельное выполнение N независимо разработанных функционально эквивалентных версий с выбором корректного выхода блоком принятия решения, как правило, на основе голосования. N-версионное программирование или мультиверсионное программирование (МВП) является программным обобщением подхода N-модульного резервирования, используемого в аппаратной отказоустойчивости [47]. Версии N дают результаты блоку принятия решения, который в данном случае является блоком голосования. Блок голосования принимает все N выходов в качестве входных данных и использует их для определения правильного или лучшего выхода, если таковой существует. Как правило, нет необходимости прерывать работу всей системы во время голосования.

Рисунок 9 иллюстрирует принцип работы модели. Данный подход также может быть использован, чтобы помочь во время тестирования отлаживать версии.

С течением времени, основанное на голосовании абсолютным большинством простое отказоустойчивое программное обеспечение с N-версиями было исследовано рядом ученых, теоретически [60] и экспериментально [61].

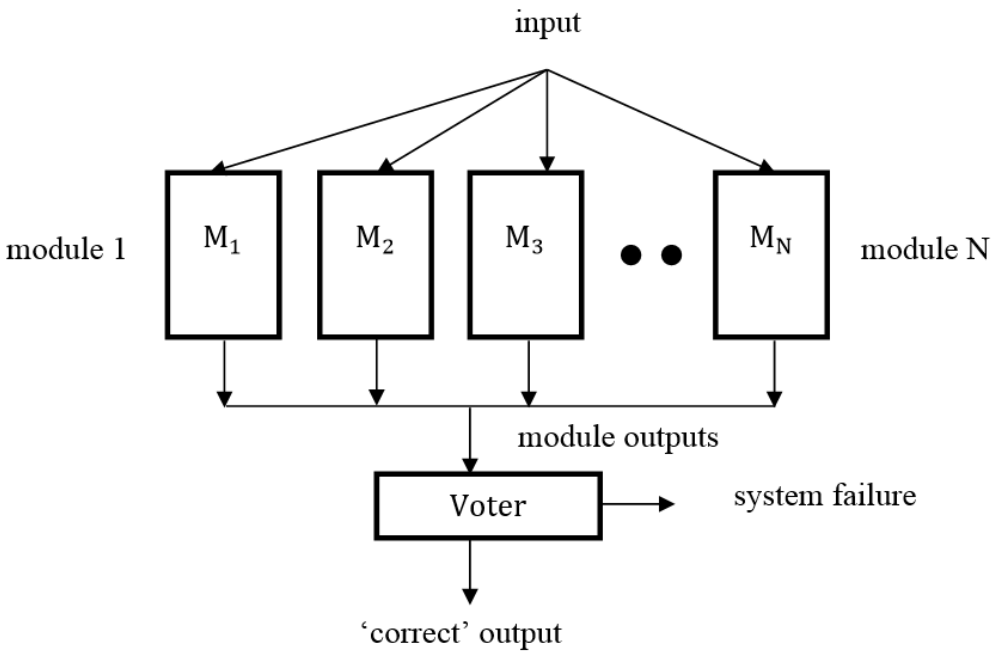


Рисунок 9 – N-версионное программирование

Разумной альтернативой голосования абсолютным большинством может стать использование результатов голосования согласованным большинством. Другим простым вариантом является медианное голосование.

Существуют варианты вышеупомянутой архитектуры для распределенных систем, некоторые из которых обсуждаются далее. Одним из таких вариантов является N – версионное программирование с самопроверкой или мультиверсионное программирование с самопроверкой (МВПС).

1.5.6 Дополнительные методы

Существуют способы объединить предыдущие простые методы для создания гибридных технологий. Изучение более продвинутых моделей, таких

как согласованные восстанавливающиеся блоки [53], голосование согласованным большинством [62] или приемочное голосование [63] являются менее распространенными и в основном теоретическими по своему характеру.

Согласованные восстанавливающиеся блоки

В [55] была предложена гибридная система, называемая согласованными восстанавливающимися блоками (СВБ). Он объединяет в себе МВП и ВБ. Если голосование в МВП не может принять решение, система возвращается к ВБ с использованием тех же модулей (могут использоваться те же результаты модуля или могут быть пересчитаны выходы модулей, если подозревается плавающий отказ). Только в том случае, если МВП и ВБ не сработали, система возвращает ошибку. Системная блок-схема приведена на рисунке 10, где блок ВБ означает, что приемочные испытания применяются к выходам вариантов в блоке МВП. Первоначально СВБ предлагалось для рассмотрения случаев множественных правильных выходов, поскольку соответствующее приемочное тестирование поможет избежать этой проблемы.

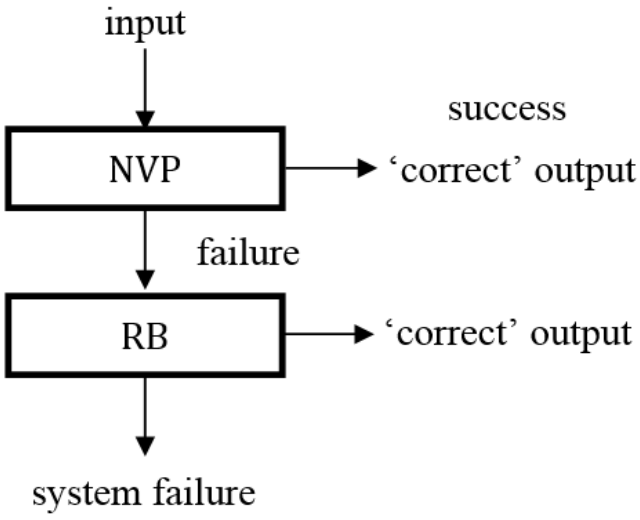


Рисунок 10 – Согласованные восстанавливающиеся блоки

Приемочное голосование

Обратная схема гибридной схемы СВБ, называемая приемочным голосованием (ПГ), была предложена [63]. Как и в МВП, все модули могут

выполняться параллельно. Результаты каждого модуля затем представляются на приемочный тест. Если приемочный тест принимает выход, то затем он передается в блок голосования. Система показана на рисунке 11

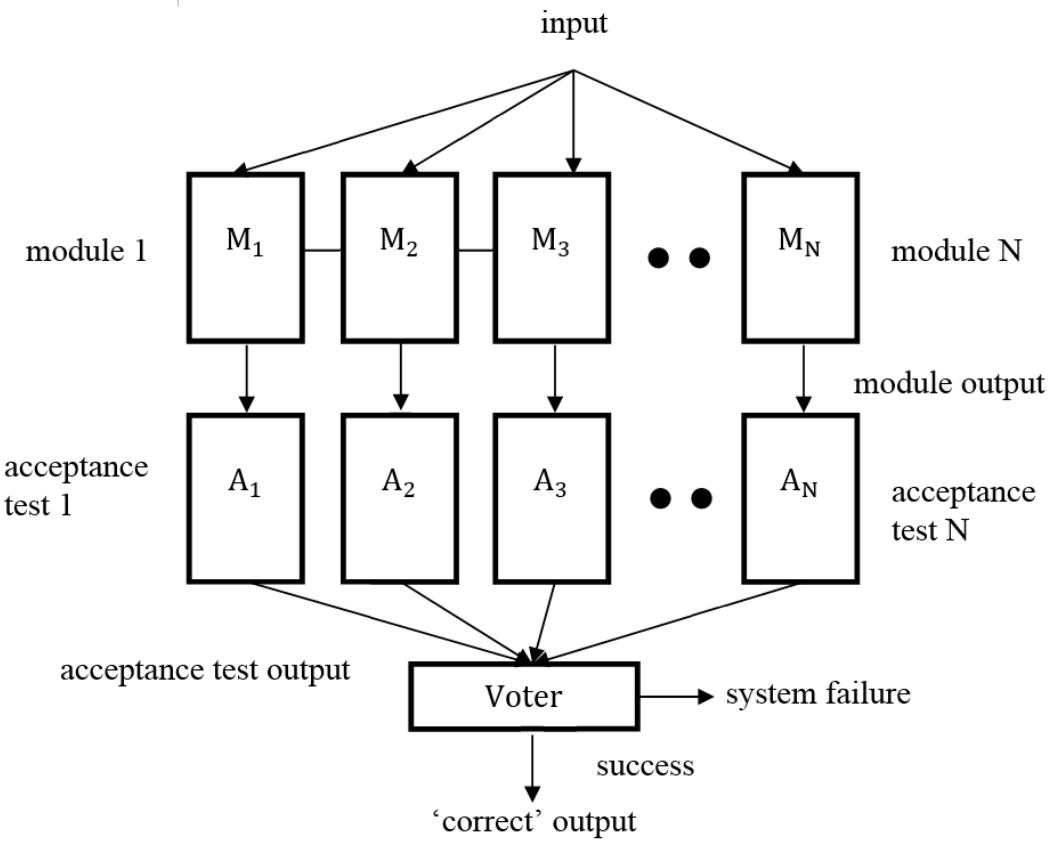


Рисунок 11 – Приемочное голосование

Блок голосования видит только те результаты, которые прошли приемочный тест. Это означает, что блок голосования не может обрабатывать одинаковое количество выходов при каждом вызове, и, следовательно, алгоритм голосования должен быть динамическим. Система не работает, если выходы не представлены блоку голосования. Если представлен только один результат, блок голосования должен считать его правильным и, следовательно, передает его на следующий этап. Только если два или более выводов согласованы, блок голосования может быть использован для принятия решения. Затем мы применяем динамическое голосование абсолютным большинством (ДГАБ) или динамическое голосование согласованным большинством (ДГСБ). Разница между ДГАБ и ГАБ заключается в том, что даже если небольшое количество результатов будет

передано блоку голосования, динамическое голосование постарается найти большинство среди них. Концепция данного голосования аналогично для ДГСБ и ГСБ.

N-версионное программирование с самопроверкой

В системе Airbus A310 используется вариант N-версионного программирования с восстановлением - МВП с самопроверкой (МВПС), [61]. В МВПС, N модулей выполняются попарно (для каждого N). Выходы из модулей можно сравнить или можно оценить на предмет правильности каким-либо другим способом. Предположим, что используется сравнение. Затем выходы каждой пары проверяются, и если они не согласуются друг с другом, ответ пары отбрасывается. Если выходы обеих версий совпадают, то эти выходы сравниваются снова. Отказ возникает, если обе пары не согласны или пары согласны, но производят разные результаты. Этот метод показан на рисунке 12 для N = 4.

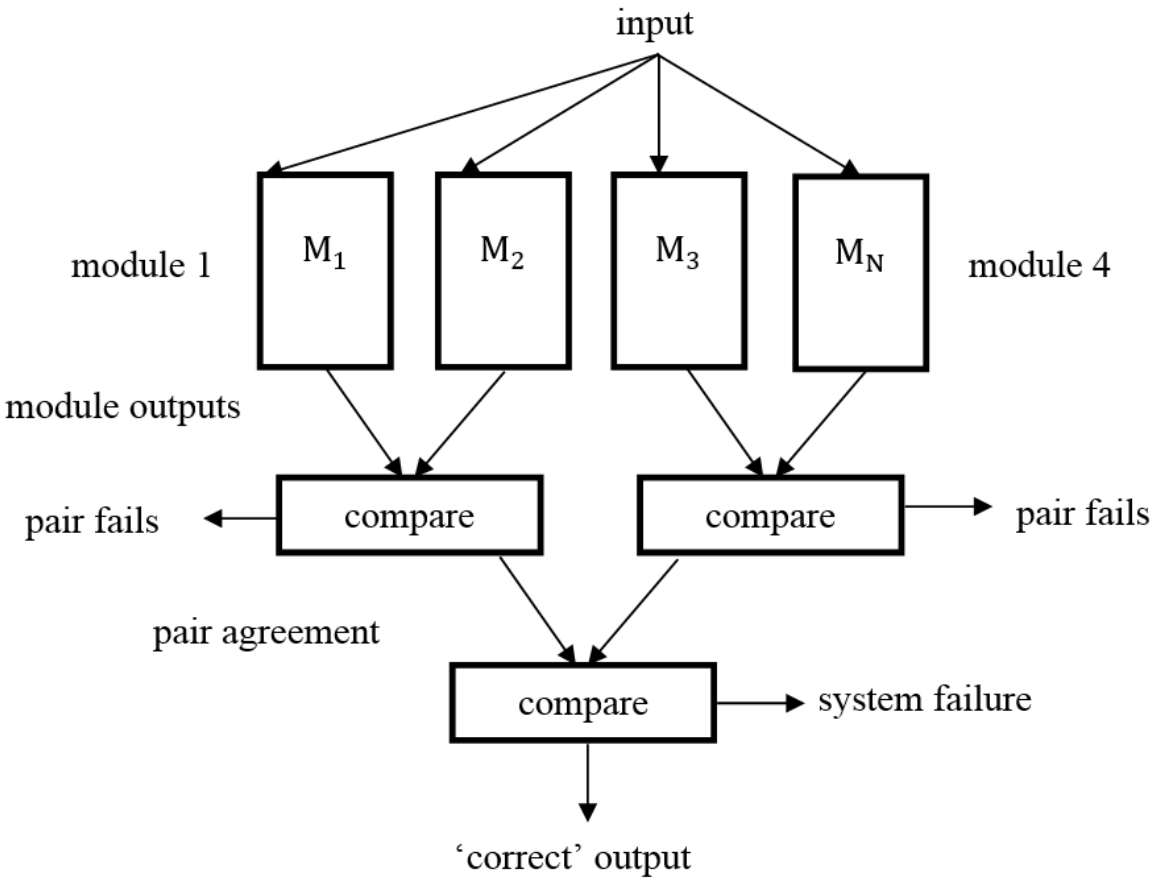


Рисунок 12 – Мультиверсионное голосование с самопроверкой (N=4)

Если сравнение выходов первой пары модулей, M1 и M2, является успешным, то выход передается на следующую фазу вычисления, и система выполняется успешно. Если эти два выхода не совпадают, тогда производится сравнение выходов второй пары модулей, M3 и M4. Если выходные сигналы второй пары совпадают, то выход передается на следующую фазу. В противном случае система возвращает ошибку. Сравнения всех четырех выходов одновременно не происходит.

t/(n-1) алгоритм

Также представляет интерес алгоритм принятия решения в мультиверсионных системах, предложенный Цзе Сюй (Jie Xu) из университета ньюкастла (University of Newcastle upon Tyne), основанный на t/(n-1) диагностируемости [64]. Для простоты будем называть его t/(n-1) алгоритмом принятия решений. Суть алгоритма не в голосовании всех выходов версий, а лишь в сравнении некоторых из них, достаточных для принятия решения [65]. Рассмотрим на примере системы с количеством версий N=5 и максимальным количеством ошибок t=2, то есть рассмотрим 2/(5-1) вариант. При количестве ошибок не превышающем t, алгоритм гарантирует выбор правильного варианта из N выходов версий, однако и при превышении количества неправильны выходов версий система не обязательно выберет неверный, с определенной вероятностью произойдет выбор правильного выхода, однако это уже не гарантируется. Рассмотрим алгоритм подробнее на заданном примере. Сравниваются попарно выходы четырех версий – 1 со 2, 2 с 3, 3 с 4, получаем три результата сравнений ω_{12}, ω_{23}, ω_{34}, равные 0, если выходы совпадают и 1, если различаются. На основании только этих трех результатов сравнения алгоритм принимает решение о переключении выхода между выходами 1, 4 и 5 версий, то есть версии 2 и 3 используются только для сравнения, значения их выходов никогда не используется в качестве выхода системы. Более наглядно схему работы можно изучить на рисунке 13.

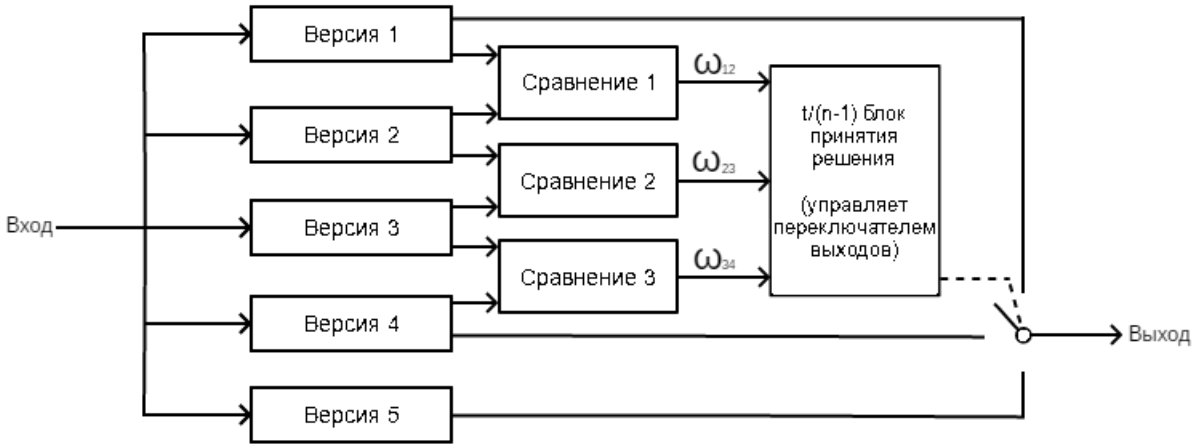


Рисунок 13 - Архитектура t/(n-1) алгоритма при n=5 и t=2

Как видно из рисунка 1, схема принятия решения в t/(n-1) алгоритме относительно не сложна, в случае пяти мультиверсий для принятия решения (правильного управления переключателем выходов) необходимы только результаты трех парных сравнений выходов четырех версий, значение выхода пятой версий для принятия решения не используется. С логикой управления переключателем выходов на основании результатов сравнений при n=5 можно ознакомиться в таблице 1.

Таблица 1. Возможные варианты выбора на основе выходов компараторов для n=5

ω_{12}	ω_{23}	ω_{34}	Предположительно правильные версии
0	0	0	1, 2, 3, 4
0	0	1	1, 2, 3
0	1	0	5
0	1	1	1, 2
1	0	0	2, 3, 4
1	0	1	5
1	1	0	3, 4
1	1	1	5

Изучив первые рисунок и таблицу, можно прийти к выводу, что, при относительно надежных версиях в большей части случаев компараторы будут возвращать (0;0;0) и на выход будет подаваться значение выполнения первой

версии. Так же можно сделать вывод об отсутствии необходимости каждый раз исполнять пятую версию, а делать это только в случае соответствующих значений результатов сравнений, когда на выход необходимо подать именно результат пятой версии. Этот факт снижает среднюю нагрузку, требуемую среде исполнения мультиверсионного ПО для работы (в подавляющем большинстве случаев будут рассчитаны 4 из 5 версий). Сам алгоритм принятия решения также является в значительной мере менее ресурсоемким по сравнению с голосованием, особенно с взвешенными его модификациями, где при каждом голосовании исполняются все версии, создаются классы и рассчитываются веса для каждого из них. Для t/(n-1) при n=5 необходимо только три простые операции сравнения с бинарным выходом. Далее осуществляется однозначный, априорно заданный выбор выхода для одного из восьми возможных сочетаний значений выходов компараторов (Таблица 1).

Выводы

На основании трудов различных исследователей рассмотрены теоретические и практические аспекты проектирования отказоустойчивых систем управления, в том числе программной составляющей критичных по надежности систем управления, надежностных характеристик программных модулей, механизмов обеспечения надежности программ с помощью введения избыточности. Изучена как классическая теория (Avizienis A.A., Lyu M.R., Boehm B.W.), в работах этих авторов содержатся фундаментальные основы программной надежности, так и последние достижения в области решения прикладных задач: работы Липаева В.В. [66-73], посвящённые надежности функционирования программ реального времени и различным методам, технологиям и инструментальным средствам разработки сложных комплексов программ реального времени; работы Ковалева И.В. [74-83], посвящённые различным мультиверсионным подходам повышения отказоустойчивости ПО и методам оценки его надежностных характеристик; работы Царева Р.Ю. [84-

93], посвящённые методам и инструментам формирования оптимального состава программных комплексов; работы Jie Xu [64,65], в которых описан принцип t/(n-1) диагностируемости и предложен t/(n-1) алгоритм принятия решения в мультиверсионных системах, основанный на данном принципе. Однако, эти работы не могут без модификаций быть применены для решения задачи получения достоверных оценок надежности создаваемых встраиваемых СУ реального времени на ранних этапах разработки, не существует реализованных программных инструментов исследования влияния характеристик отдельных модулей, их связей и структуры системы на ее общие характеристики надежности.

Проанализирован жизненный цикл разработки программной составляющей критичных по надежности систем управления, определены действия на каждом этапе, специфичные для создания отказоустойчивых систем.

Рассмотрены различные модели надежности ПО: модель Шумана, Джелинского – Моранды, Вейса, Нельсона, модели, предложенные Коркорэном и др., позволяющие изучить характеристики надежности, атрибуты, средства, нарушения надежности, сбои и отказы.

Проведено исследование методов повышения надежности программного обеспечения, в том числе анализ моноверсионных моделей и мультиверсионных, среди которых на текущий момент наиболее распространены: модель N-версионного программирования (NVP), модель N-версионного программирования с самопроверкой (NSCP), модель восстанавливающихся блоков (RB), модель согласованных восстанавливающихся блоков (CRB) и модель t/(n-1)-версионного программирования.

На основании сравнительного анализа сделан вывод о том, что наиболее эффективными являются именно мультиверсионные модели, однако среди них нет однозначно лучшей, выбор конкретной зависит от требований к

системе и характеристик составляющих ее модулей, как программных, так и аппаратных. Возникает задача обоснованного выбора модели повышения отказоустойчивости.

Для решения задачи выбора, на этапе принятия решения, необходимо сравнение различных моделей в одинаковых условиях, с обязательной возможностью исследования в рамках работающей среды исполнения отказоустойчивого ПО. Для создания имитационной среды, моделирующей работу программного комплекса, необходимо сформировать типовую структуру системы управления, включающую ее программную составляющую, аппаратные и программные интерфейсы.

2 Типовая структура встраиваемой системы управления реального времени

В данной главе рассматривается модель программного комплекса, работающего в режиме реального времени, который в данном случае представляет собой среду исполнения мультиверсионного ПО. Данный программный комплекс и будет выступать основой для создания отказоустойчивых встраиваемых систем управления реального времени.

В системах управления, работающих в автономном режиме, без возможности физического вмешательства человека (космос, глубоководные работы, работы в условиях непригодных для нахождения человека), большая часть задач управления критична к надежности и отказоустойчивости их исполнения, поскольку в случае отказа, вызванного программным сбоем, происходит потеря управляемого объекта.

Таким образом, повышение степени отказоустойчивости программного обеспечения, реализующего данные задачи, является актуальным и требует применения новых или специфичных подходов к разработке программного обеспечения, таких как введение программной избыточности, например, мультиверсионного подхода.

2.1 ОСРВ как основа встраиваемой системы управления реального времени

Рассмотрим преимущества и недостатки применения ОСРВ на современных микроконтроллерах.

Преимущества ОСРВ для микроконтроллеров (МК):

1. *Многозадачность*. ОСРВ предоставляет программисту готовый, отлаженный механизм многозадачности. Каждую задачу в простом случае можно программировать отдельно, всю работу разбить между несколькими членами команды. Не нужно заботиться о переключении между задачами, это сделает планировщик.

2. *Временная база*. Необходимо отмерять интервалы времени. ОСРВ должна иметь этот инструмент. Он позволит выполнять действия через строго выделенные интервалы времени.

3. *Обмен данными между задачами*. Для этого в ОСРВ используется очередь.

4. *Синхронизация*. Если разные задачи используют один и тот же ресурс, например, последовательный порт, то можно использовать мьютексы и критические секции. Если необходимо выполнять задачи в строгой последовательности или при наступлении определенного события, то можно использовать семафоры или сигналы для синхронизации задач.

Недостатки ОСРВ:

1. Резкое увеличение используемой памяти программ для реализации ядра (это соотношение различно в зависимости от размеров кода самой системы, исполняющейся в рамках ОС, так и от размера ядра самой ОС, в случае комплексной системы управления автономным объектом, исполняющейся на FreeRTOS с очень малыми требованиями к ресурсам, доля памяти для исполнения самого ядра ОС является незначительной, особенно по сравнению с преимуществами, которые привносит ОС).

2. Увеличение используемого ОЗУ для хранения стека каждой задачи, семафоров, очередей, мьютексов и других объектов ядра системы.

3. Задержки при переключении между задачами на сохранение контекста.

Подводя итог, можно выделить три основные функции ядра любой ОСРВ:

1. Работа планировщика, благодаря которой создается эффект параллельного выполнения нескольких задач за счет быстрого переключения между ними.

2. Переключение контекста, благодаря которому выполнение одной задачи не сказывается на остальных задачах (задачи работают независимо).

3. Временная база, основанная на системном кванте как единице измерения времени.

Проанализировав преимущества и недостатки, которые привносит ОС при использовании на микроконтроллерах, можно сделать вывод, что система управления является достаточно сложной программной системой, разработка которой будет существенно упрощена благодаря функциям, обеспечивающимся ОСРВ, а выполнение подобного ПО без использования ОС может привести к возникновению проблем при сбоях в любом программном блоке, синхронизации потоков, совместном доступе к памяти, а обеспечить стабильную работу в режиме жесткого реального времени крайне сложно. Недостатки, в виде потребления дополнительной памяти в случае сложных систем нивелируются, поскольку прикладное ПО занимает на порядок больше ресурсов, чем ОС. Можно сделать вывод, что в случае реализации на микроконтроллерах систем управления, особенно работающих в реальном времени, предпочтительно применение ОСРВ.

Сравнительный анализ и выбор ОСРВ как основы отказоустойчивой среды исполнения программных модулей встраиваемой системы управления реального времени

Резюмируя требования, сформированные в предыдущем разделе, можно формализовать следующие требования к программному комплексу системы управления:

- Отказоустойчивость самой системы;
- Кроссплатформенность;
- Реальное время;
- Возможность реализации механизмов повышения отказоустойчивости для прикладного ПО.

С учетом существенного масштаба системы и сформулированных требований предпочтительно за основу взять существующую операционную

систему реального времени (ОСРВ) с открытым исходным кодом и возможностью его модификации и использованию для собственных нужд, в том числе и коммерческих. Самостоятельная разработка ОСРВ с нуля, ее портирование на множество архитектур, отладка и тестирование, дальнейшее развитие системы потребуют очень больших материальных и временных затрат, не предоставляя значимых преимуществ перед использованием существующих систем [94].

Рассмотрим существующие ОСРВ:

FreeRTOS

Одна из самых популярных ОСРВ на сегодняшний день. Портирована на огромное количество аппаратных платформ.

Плюсы:

1) Бесплатная;

2) Портирована на большое количество аппаратных платформ;

3) Мощный функционал;

4) Есть различные библиотеки: от графики до сетевых протоколов;

5) Хорошая документация.

Минусы:

1) Сложный процесс портирования на новые аппаратные платформы, однако, на текущий момент существует множество портов на все распространенные платформы и сообществом разрабатываются порты на вновь появляющиеся.

KeilRTX

До последнего времени эта ОСРВ была коммерческой, но недавно стала открытой. Работает только на архитектуре ARM.

Плюсы:

1) Бесплатная;

2) Легко портируется на новые аппаратные платформы (в пределах архитектуры ARM);

3) Есть различные библиотеки: графика, интернет и другое.

Минусы

1) Работать вне среды Keil с ней практически невозможно;

2) Немного урезанный функционал;

3) Поддерживается только платформы ARM;

4) Как показывает опыт, проигрывает многим ОСРВ по скорости.

UC/OS

Мощная коммерческая ОСРВ.

Плюсы:

1) Огромное количество функций и библиотек;

2) Поддерживает множество аппаратных платформ.

Минусы:

1) Закрытая коммерческая система;

2) Сложна в использовании.

QNX

POSIX-совместимая операционная система реального времени, предназначенная преимущественно для встраиваемых систем. Считается одной из лучших реализаций концепции микроядерных операционных систем.

Плюсы:

1) Способна работать практически на любом современном процессоре, используемом на рынке встраиваемых систем. Среди этих платформ присутствуют семейства x86, MIPS, PowerPC, а также специализированные семейства процессоров, такие как SH-4, ARM, StrongARM и xScale.

2) Занимает лидирующую позицию среди ОС реального времени на платформе ПК.

Минусы:

1) Закрытая коммерческая система.

2) Высокая стоимость лицензии и сильная зависимость от QNX Software Systems в плане лицензирования разработанного программного обеспечения.

Проанализировав состояния развития рынка готовых операционных систем реального времени, можно сделать вывод о том, что для реализации поставленных задач наиболее подходит FreeRTOS, поскольку она портирована практически на все интересующие нас платформы, реализует необходимый нам базовый функционал, который существенно упростит разработку системы управления на нее основе, а главное – распространяется в виде открытого исходного кода, доступного для модификации (распространяется по лицензии, основанной на GNU GPL v2) и самостоятельной сборки и снабжено документацией. Так же FreeRTOS содержит функционал для обеспечения работы в режиме реального времени, требует минимум модификаций, для реализации среды исполнения мультиверсионного ПО и разработки непосредственно прикладного ПО.

2.2 Блочно-модульная структура встраиваемой системы управления реального времени

FreeRTOS — это многозадачная, мультиплатформенная, бесплатная операционная система жесткого реального времени с открытым исходным кодом. FreeRTOS была разработана компанией Real Time Engineers Ltd. специально для встраиваемых систем. На сегодняшний момент (версия FreeRTOS 6.1.0) ОС официально поддерживает 23 архитектуры и 57 платформ (в подавляющем большинстве — микроконтроллеры). Большая часть кода FreeRTOS написана на языке Си, ассемблерные вставки минимального объема применяются лишь там, где невозможно применить Си из-за специфики конкретной аппаратной платформы [95].

Работа планировщика FreeRTOS в режиме вытесняющей многозадачности имеет много общего с алгоритмом переключения потоков в

современных ОС общего назначения. Вытесняющая многозадачность предполагает, что любая выполняющаяся задача с низким приоритетом прерывается готовой к выполнению задачей с более высоким приоритетом. Как только высокоприоритетная задача выполнила свои действия, она завершает свою работу или переходит в состояние ожидания, и управление снова получает задача с низким приоритетом.

Переключение между задачами осуществляется через равные кванты времени работы планировщика, то есть высокоприоритетная задача, как только она стала готова к выполнению, ожидает окончания текущего кванта, после чего управление получает планировщик, который передает управление высокоприоритетной задаче [96].

Таким образом, время реакции FreeRTOS на внешние события в режиме вытесняющей многозадачности - не больше одного кванта времени планировщика, который можно задавать в настройках. По умолчанию он равен 1 мс. Если готовы к выполнению несколько задач с одинаковым приоритетом, то в таком случае планировщик выделяет каждой из них по одному кванту времени, по истечении которого управление получает следующая задача с таким же приоритетом, и так далее по кругу.

Кооперативная многозадачность отличается от вытесняющей тем, что планировщик самостоятельно не может прервать выполнение текущей задачи, даже если появилась готовая к выполнению задача с более высоким приоритетом. Каждая задача должна самостоятельно передать управление планировщику. Таким образом, высокоприоритетная задача будет ожидать, пока низкоприоритетная завершит свою работу и отдаст управление планировщику. Время реакции системы на внешнее событие становится неопределенным и зависит от того, как долго текущая задача будет выполняться до передачи управления. Кооперативная многозадачность применялась в семействе ОС Windows 3.x.

Вытесняющая и кооперативная концепции многозадачности объединяются вместе в гибридной многозадачности, когда вызов планировщика происходит каждый квант времени, но, в отличие от вытесняющей многозадачности, программист имеет возможность сделать это принудительно в теле задачи. Особенно полезен этот режим, когда необходимо сократить время реакции системы на прерывание. Допустим, в текущий момент выполняется низкоприоритетная задача, а высокоприоритетная ожидает наступления некоторого прерывания. Далее происходит прерывание, но по окончании работы обработчика прерываний выполнение возвращается к текущей низкоприоритетной задаче, а высокоприоритетная ожидает, пока закончится текущий квант времени.

Однако если после выполнения обработчика прерывания передать управление планировщику, то он передаст управление высокоприоритетной задаче, что позволяет значительно сократить время реакции системы на прерывание, связанное с внешним событием [97,98].

Его составляют следующие файлы:

1. tasks.c - планировщик, реализация механизма задач;

2. queue.c - реализация очередей;

3. list.c - внутренние нужды планировщика, однако функции могут использоваться и в прикладных программах;

4. croutine.c - реализация сопрограмм (может отсутствовать в случае, если сопрограммы не используются) [99].

Заголовочные файлы, которые находятся в директории Source/Include:

1. tasks.h , queue.h , list.h , croutine.h - заголовочные файлы соответственно для одноименных файлов с кодом;

2. FreeRTOS.h - содержит препроцессорные директивы для настройки компиляции;

3. mpu_wrappers.h - содержит переопределения функций программного интерфейса (API-функций) FreeRTOS для поддержки модуля защиты памяти (MPU);

4. portable.h - платформенно-зависимые настройки;

5. projdefs.h - некоторые системные определения;

6. semphr.h - определяет API -функции для работы с семафорами, которые реализованы на основе очередей;

7. StackMacros.h - содержит макросы для контроля переполнения стека. Каждая аппаратная платформа требует небольшой части кода ядра, которая реализует взаимодействие FreeRTOS с этой платформой.

Весь платформенно-зависимый код находится в поддиректории /Source/Portable, где он систематизирован по средам разработки (IAR, GCC и т. д.) и аппаратным платформам (например, AtmelSAM7S64, MSP430F449). К примеру, поддиректория /Source/Portable/ GCC/ATMega323 содержит файлы port.c и portmacro.h, реализующие сохранение/восстановление контекста задачи, инициализацию таймера для создания временной базы, инициализацию стека каждой задачи и другие аппаратно-зависимые функции для микроконтроллеров семейства mega AVR и компилятора WinAVR (GCC). Отдельно следует выделить поддиректорию /Source/Portable/MemMang, в которой содержатся файлы heap_1.c , heap_2.c , heap_3.c , реализующие 3 различных механизма вы деления памяти для нужд FreeRTOS. В директории /Demo находятся готовые к компиляции и сборке демонстрационные проекты (Demo 1 , Demo 2 , ..., Demo N). Общая часть кода для всех демонстрационных проектов выделена в поддиректорию /Demo/Common.

Чтобы использовать FreeRTOS в своем проекте, необходимо включить в него файлы исходного кода ядра и сопутствующие заголовочные файлы. Нет необходимости модифицировать их или понимать их реализацию. Например, если планируется использовать порт для микроконтроллеров MSP430 и GCC-

компилятор, то для создания проекта «с нуля» понадобятся поддиректории / Source/ Portable/GCC/MSP430F449 и /Source/Portable/ MemMang. Все остальные поддиректории из директории /Source/Portable не нужны и могут быть удалены.

Если же планируется модифицировать существующий демонстрационный проект (что, собственно, и рекомендуется сделать в начале изучения FreeRTOS), то понадобятся также поддиректории /Demo/msp430_GCC и /Demo/Common. Остальные поддиректории, находящиеся в /Demo, не нужны и могут быть удалены.

При создании приложения рекомендуется использовать makefile (или файл проекта среды разработки) от соответствующего демонстрационного проекта как отправную точку. Целесообразно исключить из сборки (build) файлы из директории /Demo, заменив их своими, а файлы из директории /Source оставить нетронутыми. Это гарантия того, что все исходные файлы ядра FreeRTOS будут включены в сборку и настройки компилятора останутся корректными.

Следует упомянуть также о заголовочном файле FreeRTOSConfig.h , который находится в каждом демонстрационном проекте [100].

FreeRTOSConfig.h содержит определения (#define), позволяющие произвести настройку ядра FreeRTOS:

1. Набор системных функций.

2. Использование сопрограмм.

3. Количество приоритетов задач и сопрограмм.

4. Размеры памяти (стека и кучи).

5. Тактовая частота МК.

6. Период работы планировщика — квант времени, выделяемый каждой задаче для выполнения, который обычно равен 1 мс. Отключение некоторых системных функций и уменьшение количества приоритетов позволяет уменьшить расход памяти программ и данных. В дистрибутив FreeRTOS

включены также средства для конвертирования трассировочной информации, полученной от планировщика, в текстовую форму (директория /TraceCon) и текст лицензии (директория /License) [101].

2.2.1 Компоненты проектируемого программного комплекса

Проектируемый программный комплекс встраиваемой СУ реального времени (рисунок 14). Основой для реализации данной системы является FreeRTOS, однако, стандартные механизмы ее функционирования позволяют выступать ей в качестве моноверсионной среды. Таким образом, требуется разработка дополнительного программного компонента - блока принятия решений, обеспечивающего как реализацию алгоритмов голосования, так и принятия решений о функционировании тех или иных мультиверсий.

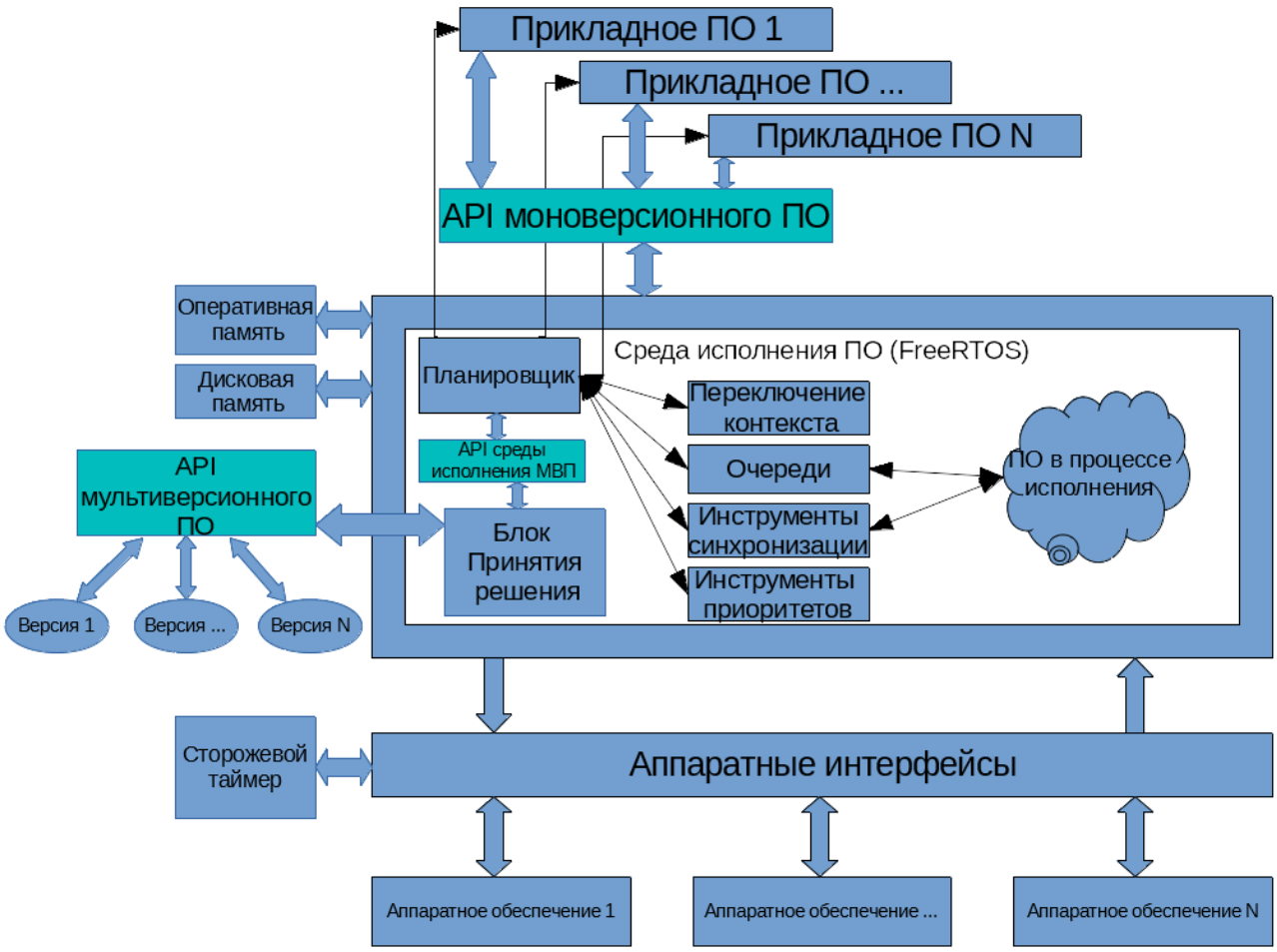


Рисунок 14 – Концептуальная структура встраиваемой СУ реального времени

2.2.2 Ядро операционной системы реального времени

FreeRTOS написана на Си с небольшим количеством ассемблерного кода (логика переключения контекста) и ее ядро представлено всего тремя Си файлами.

Основой ОСРВ является ядро операционной системы. Ядро реализует основополагающие функции любой ОС. В ОС общего назначения, таких как Windows и Linux, ядро позволяет нескольким пользователям выполнять множество программ на одном компьютере одновременно. Каждая выполняющаяся программа представляет собой задачу. Если ОС позволяет одновременно выполнять множество задач, она является мультизадачной.

Большинство процессоров могут выполнять только одну задачу в один момент времени. Однако при помощи быстрого переключения между задачами достигается эффект параллельного выполнения всех задач. На рисунке 15 показано истинно параллельное выполнение двух задач. В реальном же процессоре при работе ОСРВ выполнение задач носит периодический характер: каждая задача выполняется определенное время, после чего процессор «переключается» на следующую задачу.

Планировщик - это часть ядра ОСРВ, которая определяет, какая из задач, готовых к выполнению, выполняется в данный конкретный момент времени. Планировщик может приостанавливать, а затем снова возобновлять выполнение задачи в течение всего ее жизненного цикла (то есть с момента создания задачи до момента ее уничтожения). Политика планировщика — это алгоритм, по которому функционирует планировщик для принятия решения, какую задачу выполнять в данный момент времени. Алгоритм работы планировщика в ОС общего назначения заключается в предоставлении каждой задаче процессорного времени в равной пропорции. Алгоритм работы планировщика в ОСРВ отличается и будет описан ниже.

Среди всех задач в системе в один момент времени может выполняться только одна задача. Говорят, что она находится в состоянии выполнения.

Остальные задачи в этот момент не выполняются, ожидая, когда планировщик выделит каждой из них процессорное время. Таким образом, задача может находиться в двух основных состояниях: выполняться и не выполняться.

Кроме того, что выполнение задачи может быть приостановлено планировщиком принудительно, задача может сама приостановить свое выполнение. Это происходит в двух случаях. Первый - это когда задача «хочет» задержать свое выполнение на определенный промежуток времени (в таком случае она переходит в состояние сна (sleep)). Второй - когда задача ожидает освобождения какого-либо аппаратного ресурса (например, последовательного порта) или наступления какого-то события (event), в этом случае говорят, что задача блокирована (block). Блокированная или «спящая» задача не нуждается в процессорном времени до наступления соответствующего события или истечения определенного интервала времени. Функции измерения интервалов времени и обслуживания событий берет на себя ядро ОСРВ [102].

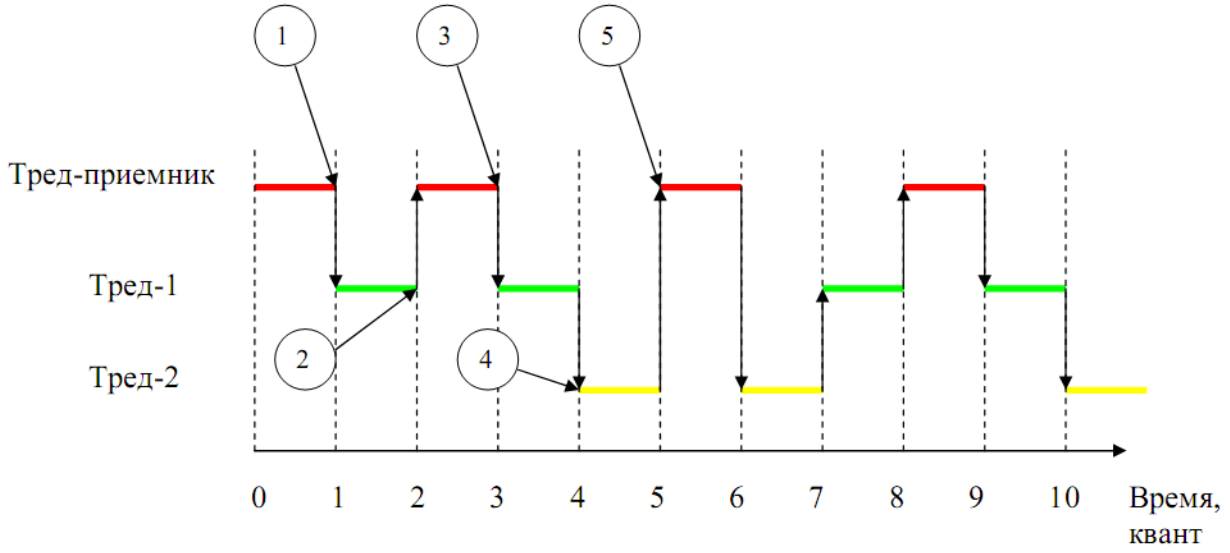


Рисунок 15 – Параллельное выполнение задач в ОСРВ

Особенности функционирования FreeRTOS.

FreeRTOS принадлежит, развивается и поддерживается Real Time Ingineers Ltd. Компания Real Time Engineers Ltd. работает в тесном партнерстве с ведущими мировыми компаниями по производству чипов, уже

более десяти лет, чтобы предоставить вам высококачественное, коммерческого уровня и совершенно бесплатное программное обеспечение.

FreeRTOS идеально подходит для встроенных приложений реального времени, которые исполняются на микроконтроллерах или небольших микропроцессорах. Этот тип приложения обычно включает в себя сочетание жестких и мягких требований реального времени.

Мягкие требования в режиме реального времени это те, которые устанавливают временной интервал, но нарушение крайнего срока не приводит к отказу системы. Например, слишком медленное реагирование на нажатия клавиш может сделать систему субъективно раздражающе невосприимчивой, не делая ее при этом непригодной для использования.
Жесткие требования в режиме реального времени это те, которые определяют временные рамки, и нарушение предельного срока приведет к абсолютному отказу системы. Например, автомобильная подушка безопасности водителя может принести больше вреда, чем пользы, если она слишком медленно реагирует на сигнал от датчика удара.

FreeRTOS это ядро реального времени (или планировщик реального времени), поверх которого встроенные приложения могут быть выполнены с удовлетворением их жестких требований в реальном времени. Он позволяет создавать приложения, как набор независимых потоков. На процессоре, имеющем только одно ядро, может выполняться только один поток в любой момент времени. Ядро решает, какой поток должен выполняться, учитывая приоритет, назначенный каждому потоку разработчиком приложения.

В простейшем случае разработчик приложения может назначать более высокие приоритеты для потоков, которые реализуют жесткие требования в реальном времени, и более низкие приоритеты для потоков, которые реализуют мягкие требования в реальном времени. Это гарантировало бы, что потоки с жестким требованием реального времени всегда выполняются перед

потоками с мягким требованием реального времени, но решения о приоритетном назначении не всегда являются упрощенными.

Существует множество хорошо зарекомендовавших себя методов написания качественного встроенного программного обеспечения без использования ядра и, если разрабатываемая система проста эти методы могут обеспечить наиболее подходящее решение. В более сложных случаях вполне вероятно, что использование ядра было бы предпочтительным, но там, где происходит точка пересечения, всегда имеет место субъективный момент.

• *Синхронизация*. Ядро ответственно за синхронизацию выполнения и обеспечивает связанный со временем API для приложения. Это позволяет упростить структуру программы приложения и уменьшить общий размер кода.

• *Эксплуатационная надёжность/расширяемость*. Уточнение деталей синхронизации приводит к меньшему количеству взаимозависимостей между модулями и позволяет программному обеспечению развиваться контролируемым и предсказуемым образом. Кроме того, ядро ответственно за синхронизацию, поэтому производительность приложения менее подвержена изменениям при использовании базового оборудования.

• *Модульность*. Задачи - это независимые модули, каждая из которых должна иметь четко определенную цель.

• *Развитие команды*. У задач должны также быть четко определенны интерфейсы, это позволит более простую разработку командами.

• *Более легкое тестирование*. Если задачи представляют собой четко определенные независимые модули с чистыми интерфейсами, они могут быть протестированы изолировано.

• *Повторное использование кода*. Большая модульность и меньшее количество взаимозависимостей приводят к такому коду, который можно повторно использовать затратив меньше усилий.

• *Повышение эффективности*. Использование ядра позволяет программному обеспечению полностью управлять событиями, поэтому время обработки не тратится впустую, путем опроса событий, которые не произошли. Код выполняется только тогда, когда есть что-то, что должно быть выполнено. Счётчик сохранения эффективности – это необходимость обработки такта прерывания RTOS и переключение исполнения из одной задачи в другую. Однако приложения, которые не используют RTOS обычно включают некоторую форму прерывания такта.

• *Время простоя*. Задача простоя создается автоматически при запуске планировщика. Он выполняется каждый раз, когда нет ни каких процессов приложения, желающих выполниться. Процесс простоя может использоваться для измерения доступной производительности обработки, проверки данных или просто перевод процессора в режим малой мощности.

• *Управление питанием*. Повышение эффективности, получаемое с помощью RTOS, позволяет процессору проводить больше времени в режиме с низким энергопотреблением. Потребляемая мощность может значительно снизиться, устанавливая процессор в состояние пониженного потребления питания каждый раз, когда выполняется процесс Idle (Простоя). FreeRTOS также имеет специальный режим без такта. Использование режима без такта позволяет процессору перейти в режим с более низким энергопотреблением, если возможно и оставаться в режиме с низким энергопотреблением дольше.

• *Гибкая обработка прерываний*. Обработки прерываний могут быть очень короткими, откладывая обработку процесса, созданного разработчиком приложения или процесс - демон FreeRTOS.

• *Требования к смешанной обработке*. Простые шаблоны проектирования могут обеспечить сочетание периодических, непрерывных и управляемых событий обработки в приложении. Кроме того, жесткие и мягкие требования в реальном времени могут быть выполняются путем выбора соответствующей задачи и приоритетов прерывания.

Возможности FreeRTOS. FreeRTOS имеет следующие стандартные функции:

• Вытесняющая или кооперативная многозадачность;

• Очень гибкое назначение приоритета задачи;

• Гибкий, быстрый и легкий механизм уведомления задачи;

• Очереди;

• Двоичные семафоры;

• Счетные семафоры;

• Мьютексы;

• Рекурсивные мьютексы;

• Таймеры программного обеспечения;

• Группы событий;

• Хук функции тактов;

• Хук функции простоя;

• Проверка переполнения стека;

• Запись трассировки;

• Сбор статистики времени выполнения задачи;

• Дополнительное коммерческое лицензирование и поддержка;

• Полная вложенная модель загрузки прерываний (для некоторых архитектур);

• Режим без тактов для экстремально низкого энергопотребления;

• Программное управление стеком прерывания, когда это необходимо (это может помочь сэкономить память) [103].

2.3Механизм очередей для межпотокового обмена данными в ОСРВ

Программы, написанные с помощью FreeRTOS представляют собой набор независимых задач – миниподпрограмм, которым требуется эффективный и потокобезопасный механизм обмена данными, в случае FreeRTOS – это очереди. Очередь – это простой буфер, организованный по принципу First In First Out (хотя также можно писать в начало очереди, а не в

конец) буфер, который может хранить фиксированное число элементов, известного размера. Запись в очередь – это побайтовое копирование данных в буфер, чтение – копирование данных и удаление из очереди.

Очереди – это, по сути, независимые объекты, которые могут иметь множество писателей, и читателей, без боязни прочитать\записать битые данные. При чтении данных, опционально, мы можем указать время, в течение которого задача должна находиться в ожидании получения новых данных. При записи данных мы также можем указать данное время, но уже для ожидания места в очереди.

Основные функции по работе с очередями в FreeRTOS: Перед использованием любой очереди, она должна быть создана. RAM память для очереди выделяется из памяти, отведенной FreeRTOS, и ее размер равен размер данных + размер структуры очереди. В коде каждая очередь представлена ее идентификатором, типа *xQueueHandle*.

xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize);

uxQueueLength – максимальное число элементов, которое очередь может хранить в единицу времени.

uxItemSize – размер каждого элемента очереди.

return xQueueHandle, или NULL – в случае если очередь создана будет возвращен соответствующий идентификатор, если нет, т.е. недостаточно памяти, то будет возвращен NULL.

Для записи в очередь используют, также специальные функции:

*portBASE_TYPE xQueueSendToFront(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait);*

*portBASE_TYPE xQueueSendToBack(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait); // Или эквивалент portBASE_TYPE xQueueSend(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait);*

xQueue – идентификатор очереди, в которую записываем данные. ***pvItemToQueue*** – указатель на элемент, который будет помещен в очередь. ***xTicksToWait*** – время, в течение которого задача должна находиться в заблокированном состоянии, чтобы появилось место в очереди. Можно указать ***portMAX_DELAY***, чтобы она находилась в блокированном состоянии в течение неопределенного времени, т.е. пока не появится место в очереди.
return pdPASS, или errQUEUE_FULL – в случае, если новый элемент успешно записан в очередь, то функция возвращает *pdPASS*, если места не достаточно, и указано время *xTicksToWait*, то задача перейдет в заблокированное время, для ожидания места в очереди.

Для чтения данных, используются 2 функции, основное отличие, которых в том, что *xQueueReceive* удаляет элемент из очереди, а xQueuePeek — нет.

*portBASE_TYPE xQueueReceive(xQueueHandle xQueue, const void * pvBuffer, portTickType xTicksToWait)*;

*portBASE_TYPE xQueuePeek(xQueueHandle xQueue, const void * pvBuffer, portTickType xTicksToWait)*;

xQueue – идентификатор очереди, в которую записываем данные. ***pvBuffer*** – указатель на буфер памяти, в который будут прочитаны данные из очереди. Тип буфера равен типу элементов очереди.
xTicksToWait – время, в течение которого задача должна находиться в заблокированном состоянии, чтобы данные появились в очереди. Можно указать ***portMAX_DELAY***, чтобы задача находилась в блокированном состоянии в течение неопределенного времени, т.е. пока не появятся новые данные в очереди. ***return pdPASS, или errQUEUE_EMPTY*** – в случае, если новый элемент успешно прочитан из очереди, то функция возвращает *pdPASS*, если очередь пуста, и указано время *xTicksToWait*, то задача перейдет в заблокированное время, для ожидания новых данных в очереди.

Для просмотра количества элементов в очереди, можно использовать функцию *unsigned portBASE_TYPE uxQueueMessagesWaiting(xQueueHandle xQueue)*;

Важно: вышерассмотренные функции, нельзя использовать в ISR (прерываниях) и для них существуют, специальные версии со специальным суффиксом **ISR**, поведение которых аналогично предыдущим функциям, за исключением последнего параметра:

*portBASE_TYPE xQueueSendToFrontFromISR(xQueueHandle xQueue, void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken);*
*portBASE_TYPE xQueueSendToBackFromISR(xQueueHandle xQueue, void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken)*;

*portBASE_TYPE xQueueReceiveFromISR(xQueueHandle xQueue, const void *pvBuffer, portBASE_TYPE *pxHigherPriorityTaskWoken)*;

pxHigherPriorityTaskWoken – так как запись в очередь может привести к разблокированию задачи, ожидающей данных и имеющей большой приоритет, чем текущая задача, то необходимо выполнить форсированное переключение контекста (для этого необходимо вызвать макрос taskYIELD()). В случае необходимости данный параметр будет равен **pdTRUE**.

Передача информации при помощи очередей возможна между любыми потоками. В то время, как для задач, решаемых одним потоком, достаточно интерфейса очередей, в системах управления возникает необходимость решения одной задачи множеством потоков. Эта необходимость возникает в случае, когда одна и та же задача должна решаться разными алгоритмами. Этим обеспечивается алгоритмическая избыточность. В случае, если алгоритм может не выполниться, при его работе может возникнуть сбой – необходимо применять альтернативный алгоритм и особый программный модуль, обеспечивающий выбор выходных данных из алгоритма, выполнившегося правильно. Такие алгоритмы называются «версии», а весь их набор – мультиверсионной задачей.

Для повышения надежности систем управления, основанных на операционных системах реального времени (ОСРВ), в частности FreeRTOS, возникает задача реализовать в ее рамках среды исполнения мультиверсионного программного обеспечения для выполнения наиболее критичных к надежности задач.

2.4 Модификация стандартных компонентов ОСРВ для разработки мультиверсионной среды исполнения

Для повышения надежности систем управления, основанных на ОСРВ, в частности FreeRTOS, возникает задача реализовать в ее рамках среду исполнения мультиверсионного ПО для выполнения наиболее критичных к надежности задач.

Для реализации функционала, обеспечивающего работоспособность среды исполнения мультиверсионного программного обеспечения (ПО) в рамках операционной системы реального времени (ОСРВ) FreeRTOS необходимо разработать отдельный компонент(модуль) блока принятия решения (далее БПР). Его функционал будет несколько шире, чем классические функции БПР, описанный в литературе, поскольку его задачей будет не только выбор правильного ответа из N версий.

Процесс БПР должен обеспечить все версии необходимыми входными данными, в случае ограничения на время ответа версий, прервать выполнение версий, не успевших дать ответ в установленный срок, непосредственно принятие решения о правильном выходе на основе коллекции выходов версий, в случае применения взвешенных алгоритмов голосования, оценка правильности выходов версий и соответствующее изменение их весов, путем помещения в их стеки 1 для версий, давших верный ответ и 0 для версий, давших ответ, не совпадающий с верным, в случае четкого голосования, или, в случае нечетких модификаций алгоритмов голосования, версий, давших ответ, отличающийся от верного более чем на заданный допуск. После этого

очистка памяти от использованных данных и завершение процессов версий, если они более не нужны (не будут исполняться снова).

В связи с введение функционала среды мультиверсионного исполнения появляется необходимость формализовать еще два программных интерфейса, помимо стандартного API моноверсионного прикладного ПО, это интерфейс между версиями и БПР, который, по большей части, должен быть описан в V-spec (спецификация на разработку версий различными разработчиками, обеспечивающая гарантию их работоспособности в целевой среде исполнения), и интерфейс между БПР и планировщиком ОСРВ, поскольку БПР должен иметь приоритет выше других процессов и право на запуск, приостановку и завершение работы процессов версий.

Во FreeRTOS для дальнейшего управления процессом, необходимо знать handle (идентификатор процесса), который присваивается при его создании. Поэтому рациональнее запускать процессы версий именно из БПР, сохраняя в локальные переменные их идентификаторы, что позволяет БПР дальше управлять созданными процессами, используя стандартные команды FreeRTOS, передавая в них идентификатор нужного процесса в качестве параметра.

Блок принятия решений для реализации разработанных алгоритмов голосования

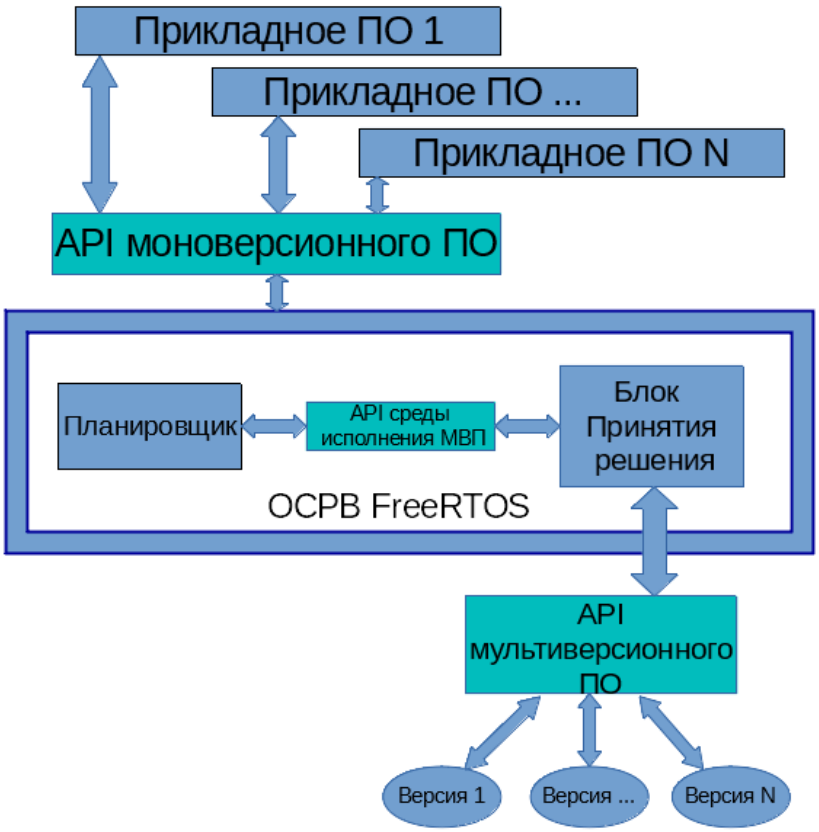


Рисунок 16 - Структура взаимодействия различных программных структур в рамках системы управления

На рисунке 16 можно наблюдать диаграмму, показывающую этапы процесса мультиверсионного исполнения прикладной задачи. После запуска задачи БПР осуществляет проверку наличия необходимых входных данных, также их корректность (тип данных, размерность, попадание в заданный диапазон и т.д., в зависимости от задачи), если входные данные проходят проверку, БПР запускает потоки версий и передает им правильные входные данные, ожидает ответа всех версий (проверяя очередь ответов), если существует ограничение на время ответа версий, то по истечению этого времени, если ответ дали не все версии, то принятие решения производится на основании полученных ответов, а процесс не успевшей дать ответ версии завершается принудительно.

В зависимости от выбранного алгоритма принятия решения для работы ему может потребоваться не менее определенного количества ответов, при сочетании ограничения на время ответа версий и алгоритма, требующего для своей работы большей части ответов версий, может произойти ситуация, когда получено меньше ответов, чем необходимо алгоритму для работы. В таком случае БПР либо возвращает ошибку, сообщая о невозможности принять решение, либо на выход отправляется выход любой из версий, успевших дать ответ, в зависимости от реализации БПР.

Однако и при наличии всех ответов версий не всегда возможно выбрать единственно верный вариант, у многих алгоритмов принятия решения существуют ограничения, например – для алгоритма голосования абсолютным большинством необходимо, чтобы совпали более чем половины версий. Если принятие решения на основе набора значений выходов возможно, оно принимается, если нет, то БПР либо возвращает ошибку, сообщая о невозможности принять решение, либо на выход отправляется выход любой из версий, в зависимости от реализации БПР. После определения правильного выхода происходит изменение весов версий в соответствии с совпадением их ответов с правильным.

Для реализации функционала, обеспечивающего работоспособность среды исполнения мультиверсионного программного обеспечения в рамках ОСРВ FreeRTOS необходимо разработать отдельный компонент блока принятия решения (далее БПР). Его функционал будет несколько шире, чем классические функции БПР, описанный в литературе, поскольку его задачей будет не только выбор правильного ответа из N версий.

Процесс БПР должен:

– обеспечить все версии необходимыми входными данными;

– в случае ограничения на время ответа версий, прервать выполнение версий, не успевших дать ответ в установленный срок;

– непосредственно принятие решения о правильном выходе на основе коллекции выходов версий;

– в случае применения взвешенных алгоритмов голосования, оценка правильности выходов версий и соответствующее изменение их весов, путем помещения в их стеки весов «1» для версий, давших верный ответ и «0» для версий, давших ответ, не совпадающий с верным, в случае четкого голосования, или, в случае нечетких модификаций алгоритмов принятия решения (голосования), версий, давших ответ, отличающийся от верного более чем на заданный допуск;

– произвести очистку памяти от использованных данных и завершить процессы версий, если они более не нужны (не будут исполняться снова).

2.5 Определение типовых программных интерфейсов в архитектуре встраиваемых систем управления реального времени

В связи с введение функционала среды мультиверсионного исполнения появляется необходимость формализовать еще два программных интерфейса, помимо стандартного API моноверсионного прикладного программного обеспечения, это интерфейс между версиями и БПР (рисунок 17), который, по большей части, должен быть описан в V-spec (спецификация на разработку версий различными разработчиками, обеспечивающая гарантию их работоспособности в целевой среде исполнения), и интерфейс между БПР и планировщиком ОСРВ, поскольку БПР должен иметь приоритет выше других процессов и право на запуск, приостановку и завершение работы процессов версий.

Во FreeRTOS для дальнейшего управления процессом, необходимо знать handle (идентификатор процесса), который присваивается при его создании. Поэтому рациональнее запускать процессы версий именно из БПР, сохраняя в локальные переменные их идентификаторы, что позволяет БПР дальше управлять созданными процессами, используя стандартные команды

FreeRTOS, передавая в них идентификатор нужного процесса в качестве параметра.

На рисунке 17 представлен архитектурный базис, формализованный средствами лингвистического обеспечения UML, отражающий принципы построения кроссплатформенного БПО и типовые программные интерфейсы.

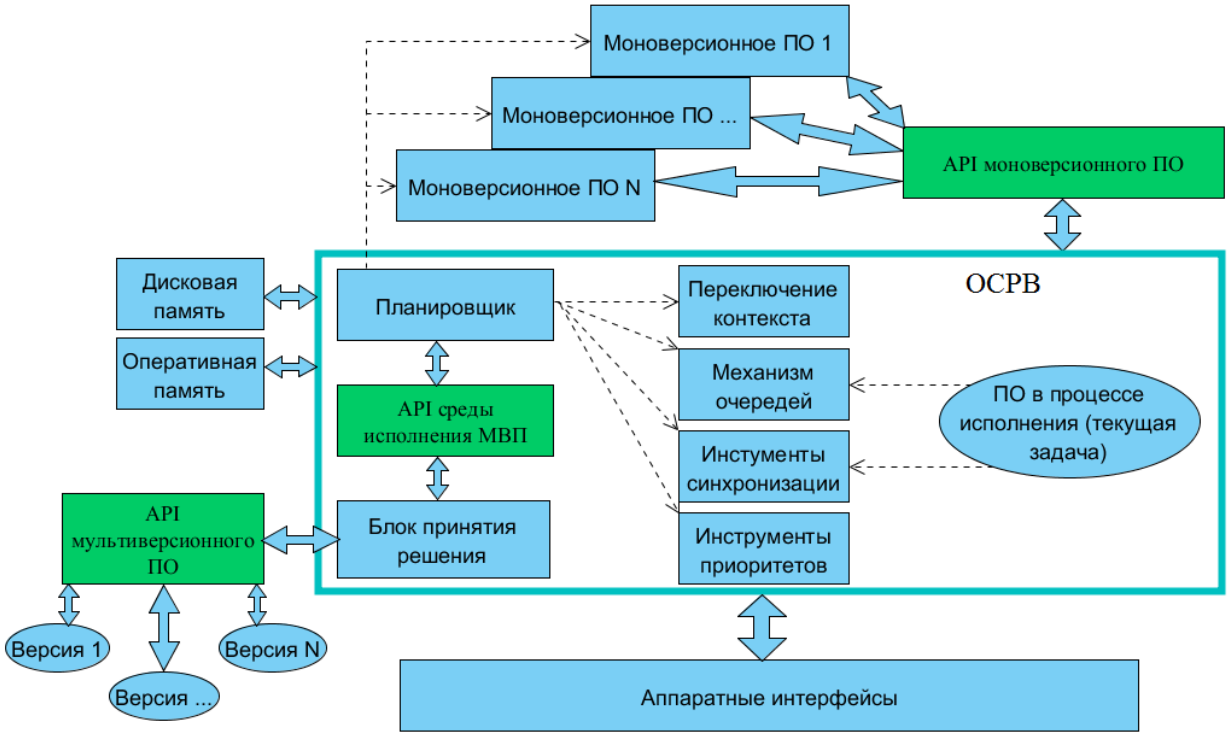


Рисунок 17 – Структура программной системы и программных интерфейсов в ней, представленные средствами UML

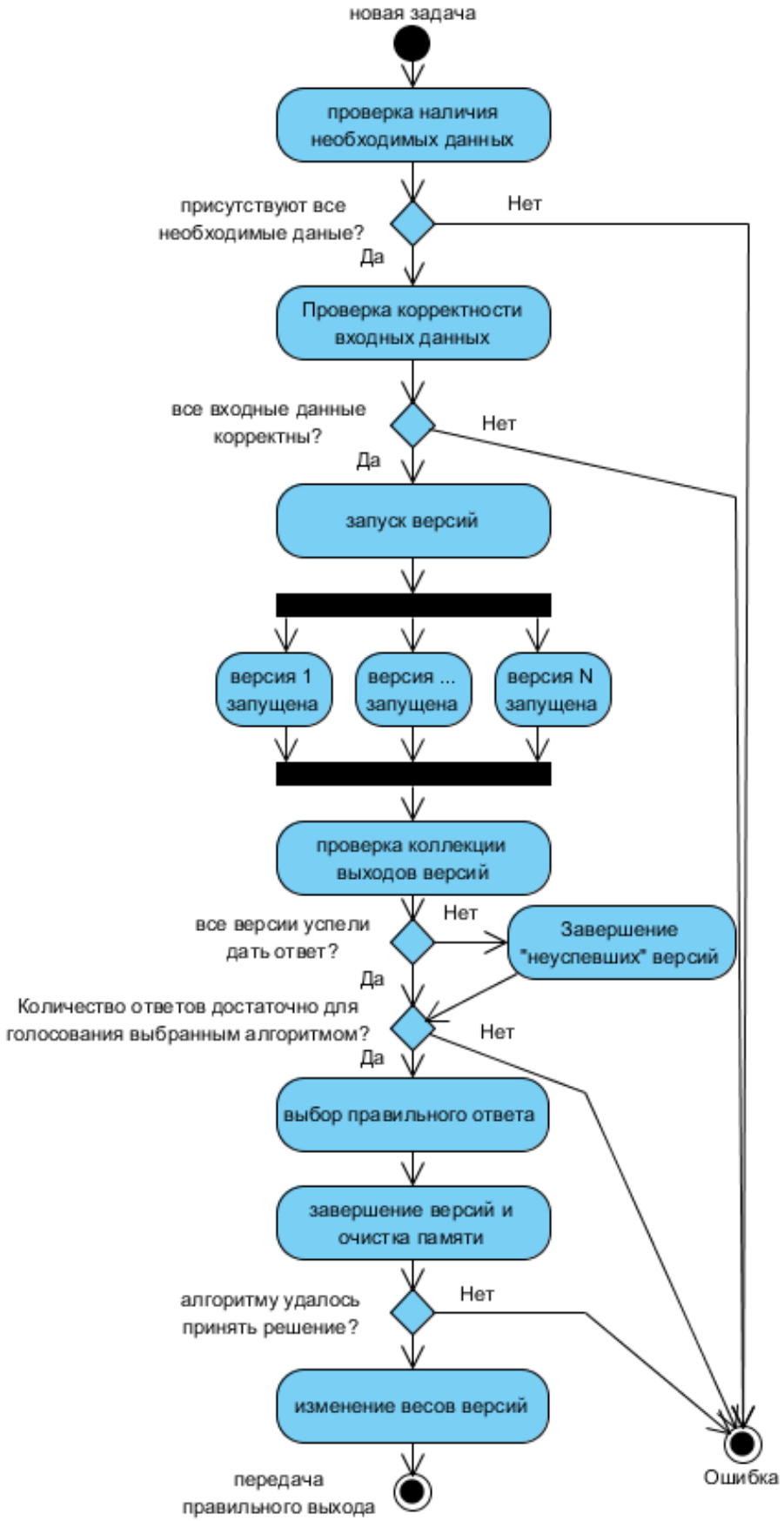


Рисунок 18 – UML-диаграмма активности, показывающая процесс мультиверсионного исполнения прикладной задачи в рамках ОСРВ

На рисунке 18 можно наблюдать UML-диаграмму активности, показывающую этапы процесса мультиверсионного исполнения прикладной задачи. После запуска задачи БПР осуществляет проверку наличия необходимых входных данных, также их корректность (тип данных, размерность, попадание в заданный диапазон и т.д., в зависимости от задачи), если входные данные проходят проверку, БПР запускает потоки версий и передает им правильные входные данные, ожидает ответа всех версий (проверяя очередь ответов).

При существующем ограничении на время ответа версий, по истечению этого времени, если ответ дали не все версии, то принятие решения производится на основании полученных ответов, а процесс не успевшей дать ответ версии завершается принудительно. В зависимости от выбранного алгоритма принятия решения для работы ему может потребоваться не менее определенного количества ответов, при сочетании ограничения на время ответа версий и алгоритма, требующего для своей работы большей части ответов версий, может произойти ситуация, когда получено меньше ответов, чем необходимо алгоритму для работы. В таком случае БПР либо возвращает ошибку, сообщая о невозможности принять решение, либо на выход отправляется ответ любой из версий, успевших дать ответ, в зависимости от реализации БПР.

Однако и при наличии всех ответов версий не всегда возможно выбрать единственно верный вариант, у многих алгоритмов принятия решения существуют ограничения, например – для алгоритма голосования абсолютным большинством необходимо, чтобы совпали ответы более, чем у половины версий. Если принятие решения на основе набора значений выходов возможно, оно принимается, если нет, то БПР либо возвращает ошибку, сообщая о невозможности принять решение, либо на выход отправляется ответ любой из версий, в зависимости от реализации БПР. После определения

правильного выхода происходит изменение весов версий в соответствии с совпадением их ответов с правильным.

Для безопасного и корректного обмена информацией между потоками FreeRTOS используется механизм очередей. Очередь – это простой FIFO буфер (хотя также существует возможность записи и в начало очереди), который может хранить число элементов, не превышающее заданного размера очереди. Запись в очередь – это побайтовое копирование данных в буфер, чтение – это копирование данных с возможностью их удаления из очереди. С точки зрения функционирования программной системы, очереди – это независимые объекты, в которые могут производить операции чтения / записи множество процессов без риска прочитать / записать повреждённые данные.

Создаются очереди функцией, имеющей вид *xQueueCreate()*. С точки зрения функционирования создаваемых API следует отметить следующие важные параметры функции: *uxQueueLength* - длинна очереди; *uxItemSize* - размер памяти, отводимый на каждый объект; *xQueueHandle* - идентификатор очереди, который позволяет обращаться к конкретной очереди.

Функции чтения элемента очереди имеют следующие варианты: *xQueueReceive()* - удаляет прочитанный элемент из очереди; *xQueuePeek()* - оставляет прочитанный элемент в очереди.

Так как написание конкретных версий может быть поручено сторонним разработчикам, для диверсификации программных версий, с целью исключения межверсионных ошибок, необходимо обеспечить передачу идентификаторов очередей входных и выходных данных в версии для их обработки.

Данная задача выполняется блоком принятия решений (БПР). Создаётся особая очередь идентификаторов очереди входных данных *BPRInQueue*. При вызове мультиверсионной функции в неё помещается идентификатор той очереди, откуда версии должны получать исходные данные. БПР запускается и считывает идентификатор из очереди идентификаторов. Из очереди с

данным идентификатором БПР получает исходные данные и анализирует их. Далее данный идентификатор передаётся каждой версии в качестве аргумента *pvParameters* и используется ею для получения исходных данных. Результат работы версии должен быть записан в очередь обработки данных, используемую БПР для выбора правильного значения результата. Следовательно, БПР также должен передавать в качестве параметра её идентификатор.

Аналогичным способом БПР получает идентификатор очереди, в которую он должен записать выходные данные версии, давшей правильный результат. Имеется очередь идентификаторов выходной очереди *BPROutQueue*, из которой БПР берёт идентификатор выходной очереди и записывает выходные данные в очередь, соответствующую ему. Так как очереди становятся доступны всем потокам при указании соответствующего идентификатора, следует ввести механизм контроля, предотвращающий изъятия элемента из очереди, не соответствующей алгоритму работы. Следует контролировать, чтобы версии программного обеспечения не вызывали функцию изъятия элемента *xQueueReceive* из очередей входных, выходных данных или очередей идентификаторов.

Перед компиляцией кода программных версий следует выполнить проверку содержимого кода, отслеживая применение команды *xQueueReceive* и избавляясь от случаев неуместного её применения.

Другой задачей БПР будет запуск отдельных версий на исполнение и контроль их исполнения. Предварительно все версии создаются в виде обычных функций на языке С.

Запуск потока версии выполняется вызовом функции *portBASE_TYPE xTaskCreate()*, имеющей следующие важные параметры: *vTaskCode* - указатель на функцию, описывающую задачи; *usStackDepth* - глубина стека функции; *pvParameters* - указатель на параметры, которые использует функция;

uxPriority - приоритет задачи; *pvCreatedTask* - идентификатор потока, используемый для дальнейшего управления создаваемым потоком.

Завершается поток функцией *vTaskDelete()*, параметром которой выступает *xTaskHandle* – идентификатор потока. Если он не указан – завершается поток, вызвавшая данную функцию.

Для управления временем выполнения задач применяются функции управления планировщиком, к примеру *vTaskDelay()*, задерживающая выполнение задачи на определенный интервал времени (параметр – *xTicksToDelay*) или *vTaskSuspend()*, приостанавливающая задачу с идентификатором (параметр – *pxTaskToSuspend*) до подачи команды *vTaskResume()*, соответствующим идентификатором в качестве параметра.

Для наглядного объяснения взаимодействия с предложенными программными интерфейсами рассмотрим UML-диаграмму последовательностей, отображающую пример очередности действий при мультиверсионном исполнении задачи с количеством версий N=3.

В процессе межкомпонентного взаимодействия участвуют следующие акторы: планировщик – стандартный элемент FreeRTOS, отвечающий за переключение между процессами; блок принятия решений – разработанный нами модуль с целью выполнения версий и принятия верного решения на основе коллекции их выходов; N-версий – функционально эквивалентные, но алгоритмически разные версии очередь исходных данных – очередь содержащая идентификатор другой очереди, в которой хранятся необходимые входные данные для решения задачи версиями; очередь обработки данных – очередь, в которую все версии помещают свои ответы в виде структуры, содержащей ответ и номер версии, необходимые БПР для принятия решения; очередь выходных данных - очередь содержащая идентификатор другой очереди, в которую необходимо поместить правильный ответ.

Следует отметить, что в случае применения невзвешенных алгоритмов принятия решения в очередь обработки данных необходимо поместить только

ответы, причем последовательность не важна, поскольку не имеет значение какой ответ принадлежит каждой из версий. Однако, в случае применения взвешенных алгоритмов нам необходимо знать какая версия дала какой ответ, поскольку у ответов будут разные веса, влияющие на изменения весов классов. Также это необходимо знать для изменения весов версий по результату принятия решения при сравнении выигравшего ответа с ответами данными версиями.

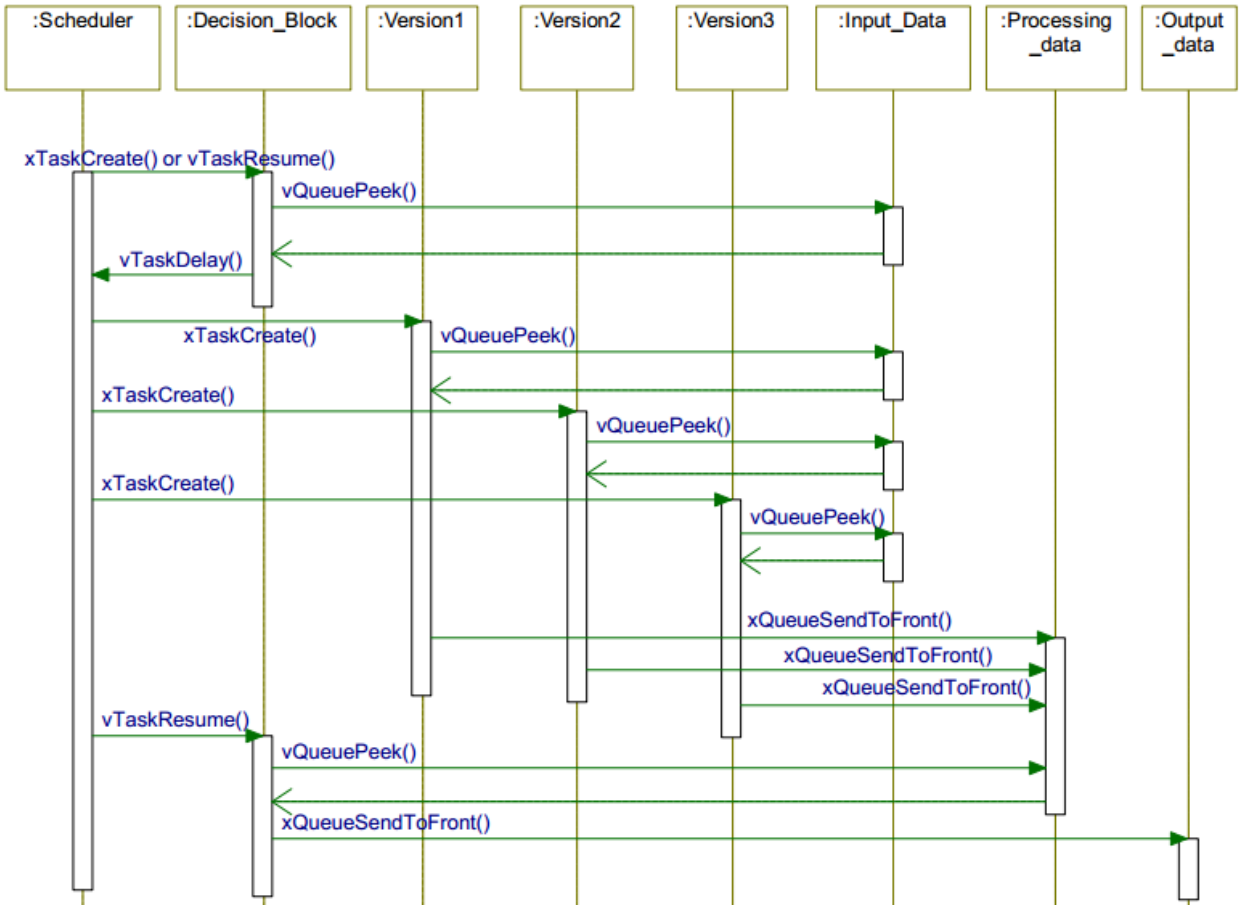


Рисунок 19 – Диаграмма последовательности процесса мультиверсионного исполнения прикладной задачи при успешном завершении всех версий

Рассмотрим процесс мультиверсионного исполнения прикладной задачи, изображенный в виде UML-диаграммы последовательности на рисунке 19. Каждая задача, выполняемая мультиверсионно, запускается из тела основной программы в соответствии с алгоритмом программного обеспечения.

1. Планировщик запускает БПР функцией *xTaskCreate()*, если это первый запуск, или передает ему управление функцией *vTaskResume()*, если он уже был запущен ранее и ожидал появление элемента в очереди исходных данных.

2. БПР обращается к очереди исходных данных, функцией *vQueuePeek()* считывая оттуда идентификатор и получая исходные данные для проверки пригодности для обработки, определяет, достаточно ли данных, их тип, диапазон значений, если данные подходят для выполнения задачи, переходит к следующему шагу.

3. БПР вызывает функцию *xTaskCreate()* для соответствующих версий, планировщик запускает потоки версий ПО.

4. Сам БПР уходит в задержку, вызывая функцию *vTaskDelay()* на интервал времени, равный отведённому для работы версий в случае ограничения на время исполнения, или “засыпает”, ожидая появления результатов в очереди обработки данных.

5. Версии начинают свою работу, получая исходные данные из очереди исходных данных функцией *vQueuePeek()*.

6. Отработав и получив результирующие данные, каждая версия функцией *xQueueSendToFront()* посылает их в очередь обработки данных;

7. По истечении времени задержки планировщик передаёт управление БПР, БПР принимает данные из очереди обработки, оценивает их и на основании алгоритма принятия решения выбирает, какие результирующие данные являются правильными

8. Принятые правильные выходные данные записываются в очередь выходных данных

9. В случае применения взвешенных алгоритмов принятия решения БПР записывает соответствующие значения в стеки весов версий.

Также рассмотрим случай, в котором не все версии успели дать ответ в установленный отрезок времени, в нашем случае – вторая версия (рисунок 20).

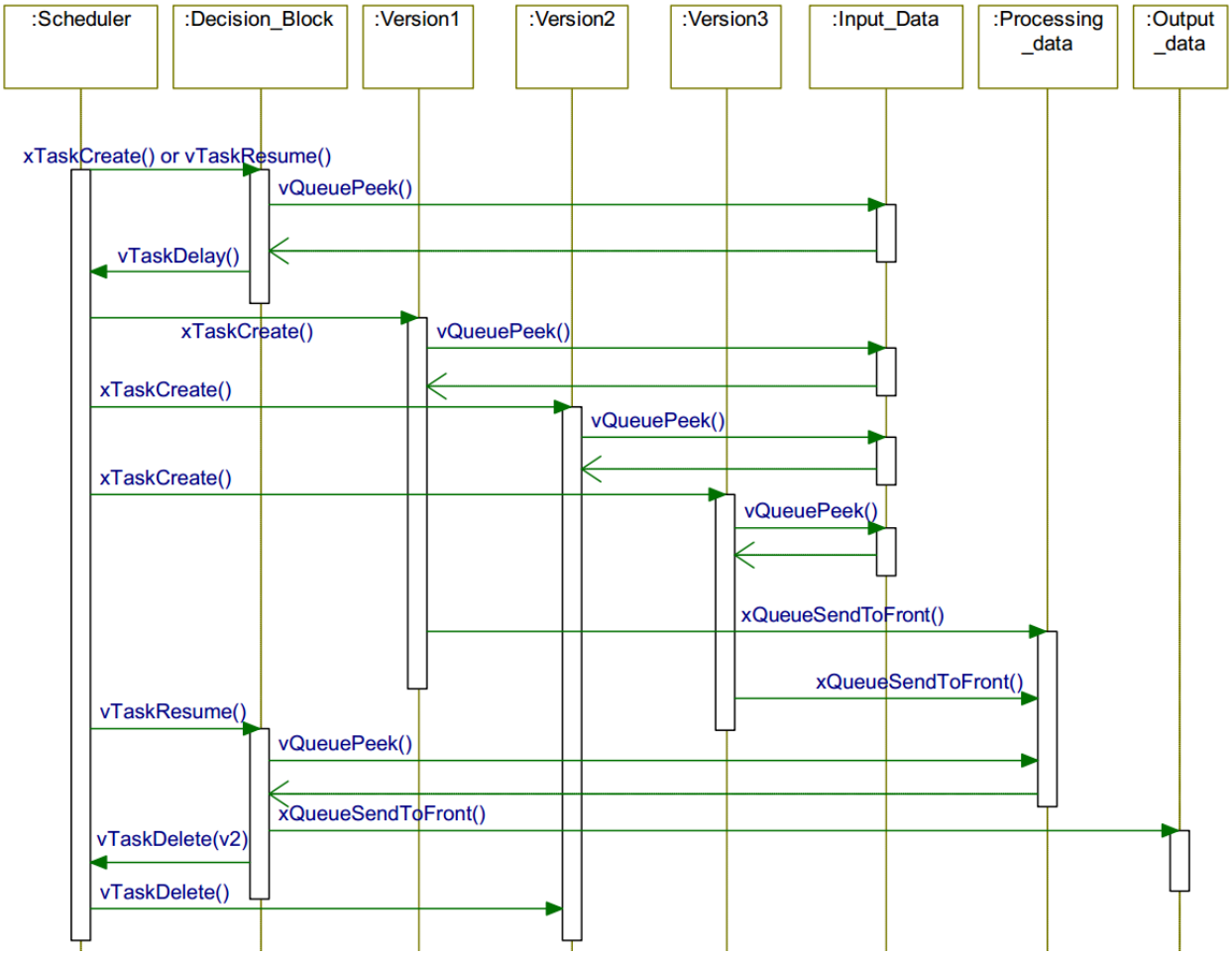


Рисунок 20 – Диаграмма последовательности процесса мультиверсионного исполнения прикладной задачи при одной версии, не успевшей дать ответ

Выводы

Во второй главе рассмотрена модель встраиваемой системы управления реального времени и ее программной составляющей. Проведен анализ необходимости применения ОС на целевых аппаратных платформах, сделан вывод о предпочтительности использования в качестве основы среды исполнения существующей операционной системы реального времени (ОСРВ) с открытым исходным кодом и возможностью его модификации. Это решение обеспечит систему многозадачностью, временной базой, механизмом обмена данными между задачами (очереди), механизмами синхронизации, работой в реальном времени. Недостатки применения ОСРВ менее

существенны и заключены, в основном, в увеличении требований к объему ОЗУ.

Рассмотрены существующие ОСРВ: FreeRTOS, KeilRTX, UC/OS, QNX. В результате анализа сделан выбор в пользу ОСРВ FreeRTOS, поскольку она обладает всем необходимым функционалом, портирована на большинство аппаратных платформ, распространяется по лицензии, разрешающей модификацию исходного кода и дальнейшее использование, в том числе и в коммерческих целях.

Проанализирована логика работы выбранной в качестве основы ОСРВ, сформированы необходимые модули и программные интерфейсы (API) для обеспечения работы как моноверсионного, так и мультиверсионного прикладного ПО.

В результате сформирована блочно-модульная структура типового программного комплекса встраиваемой системы управления реального времени.

Определено, что для реализации требуемой функциональности среды исполнения мультиверсионного ПО необходимо разработать блок принятия решения (БПР), который должен обеспечить все версии необходимыми входными данными, в случае ограничения на время ответа версий, прервать выполнение версий, не успевших дать ответ в установленный срок, непосредственно принять решение о правильном выходе на основе коллекции выходов версий и т.д.

Формализованы три программных интерфейса для обеспечения требуемой функциональности системы. Сформированы необходимые модификации планировщика для корректного исполнения мультиверсий.

В результате возникает задача выбора применяемых в блоке принятия решения моделей повышения отказоустойчивости ПО.

3 Комбинированный селективный алгоритм компоновки состава мультиверсионного программного комплекса

Основной целью данного алгоритма является достижение заданных надежностных характеристик проектируемых программных модулей – компонентов встраиваемой системы управления реального времени.

Селективный алгоритм осуществляет численный анализ выражения, в соответствии с конечной формулой для каждой модели пула, и позволяет оценить вероятность отказа каждой модели в соответствии с заданными надежностными характеристиками программного модуля, тем самым предоставляя разработчику программного компонента возможность оценки согласованности надежностных характеристик программного модуля и системы в целом.

В основу пула положены следующие модели мультиверсионной методологии: модель восстанавливающихся блоков, модель мультиверсионного программирования и модель мультиверсионного программирования с самопроверкой. Количество моделей в пуле может быть не ограничено.

Селективный алгоритм основан на модели «дерева сбоев» и модели корректности, с помощью которых вычисляется верхняя и нижняя границы надежности.

3.1 Модель «дерева сбоев»

«Дерево сбоев» состоит из нежелательного верхнего события (сбоя системы или подсистемы), связанного с основными событиями через логические затворы. Верхнее событие разделяется на составляющие его причины, связанные логическими затворами «И» и «ИЛИ», которые далее разрешаются до тех пор, пока не будут определены основные события. Основные события выражают базовые причины сбоя и представляют собой предел разрешения «дерева сбоев». Таким образом, руководствуясь

принципами методологии «дерева сбоев», формируется иерархическая логическая модель СУ с целью объявления событий сбоя, связанных с функционированием как программного, так и аппаратного обеспечения СУ.

Хотя деревья сбоев традиционно использовались для анализа аппаратных систем, они также могут выступить как вспомогательное средство проектирования для программного обеспечения. В некотором смысле анализ дерева сбоев программной системы является дополнением к формальному обзору разработки. Формальный обзор разработки помогает убедиться, что ПО делает то, что должно делать. Анализ дерева сбоев помогает гарантировать, что ПО не выполняет то, что оно не должно делать.

При анализе программного обеспечения возможными причинами сбоев могут быть комбинации ошибок программного обеспечения, входов и режимов работы. Как и при анализе механической или аппаратной системы, нежелательное событие разлагается на составляющие его причины, которые становятся основными событиями дерева сбоев. При анализе аппаратных систем основные события обычно связаны с отказами физических компонентов. В программной системе основные события представляют собой программные модули, которые могут привести к неправильному результату или принять неверный ввод. Или базовое событие может представлять неправильную установку параметра инициализации пользователем или переполнение буфера.

В дополнение к детальному анализу критического программного обеспечения с одной версией, модели дерева сбоев полезны для анализа режимов сбоев, связанных с отказоустойчивыми программными приложениями.

Модели деревьев сбоев используются для описания комбинаций событий сбоя (аппаратное и программное обеспечение), которые могут объединиться для получения недопустимого результата. Этот анализ будет игнорировать сбои в работе общих служб платформы (сети связи,

операционной системы, драйверов устройств и т. д.) и сосредоточится на отказоустойчивости прикладного программного обеспечения.

Формирование атрибутивных отличий

Особенности построения деревьев сбоев, а также их последующего анализа связаны с определением отдельных событий и их множеств, способных влиять на сбой системы в целом.

Каждая из моделей дерева сбоев будет использовать следующие обозначения для основных событий.

V# - несвязанная ошибка версии (где # - целое число от 1 до n (n - число версий));

D - независимая ошибка в решении (приемочный тест, мажоритарный избиратель, компаратор, арбитр);

RV## - связанная ошибка, которая возникает в двух разных версиях, в результате чего оба результата приводят к совпадающему ошибочному результату;

RALL - связанная ошибка, влияющая на все версии, а также на решение, вызванное несовершенными спецификациями;

H# - аппаратная ошибка, которая влияет на корректное исполнение ПО;

Pi – вероятность события i, Qi =(1- Pi).

По результату формирования атрибутивных отличий становится возможным формализовать полученную модель дерева сбоев СУ, путем определения для каждой модели конечной формулы - выражения, с целью оценки вероятности того, что будет получен недопустимый результат – вероятность сбоя модели целиком.

3.1.1 Модель «дерева сбоев» для системы восстанавливающихся блоков

Существует, по крайней мере, два разных способа сочетать аппаратное резервирование с подходом восстановления блоков к отказоустойчивости программного обеспечения. В [104] архитектура RB/1/1 дублирует блок

восстановления на двух аппаратных компонентах. Оба аппаратных компонента выполняют один и тот же вариант, а аппаратные сбои определяются путем сравнения приемочного теста и результатов вычислений. Распределенный блок восстановления (distributed recovery block DRB), выполняет различные альтернативы на разных аппаратных компонентах, чтобы улучшить производительность в случае обнаружения ошибки. В системе DRB один процессор выполняет первичное чередование, в то время как другой выполняет вторичное. Если в первичных результатах обнаружена ошибка, результаты из вторичного доступны немедленно. Модель дерева сбоев обеих систем идентична.

Модель дерева сбоев DRB показана на рисунке 21. Вычисление одной задачи приведет к недопустимым результатам, если произойдет одно из трех событий. Во-первых, если и первичный, и вторичный сбой на одном и том же входе из-за двух несвязанных ошибок или одной связанной ошибки. Во-вторых, если оба компонента оборудования испытывают сбои, то размещаемые вычисления будут нарушены и не смогут получить правильные результаты. В-третьих, если разработчик не может либо обнаружить неприемлемые результаты, либо принять правильные результаты, то вычисление не выполняется.

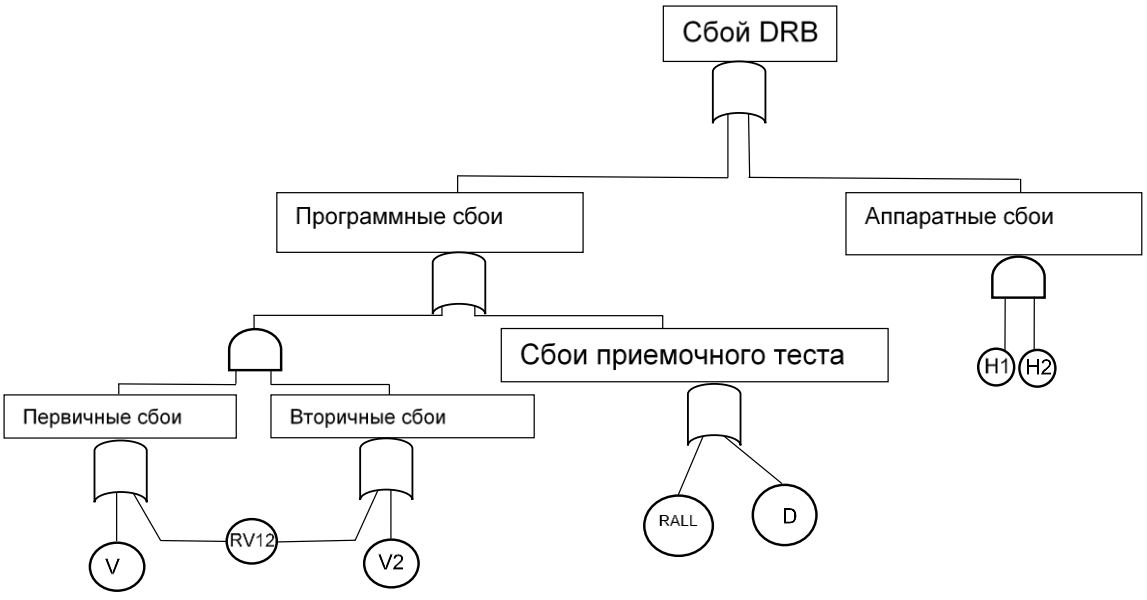


Рисунок 21 - Модель дерева сбоев для системы восстанавливающихся блоков.

Модель дерева сбоев предоставляет компактный формат для описания эффектов как программных, так и аппаратных сбоев. Даже не создавая сокращений для системы DRB, можно легко визуализировать эффекты сбоя от решения или связанной ошибки между версиями. Для системы DRB предусмотрено пять минимальных сокращений. Три из этих сечения имеют один элемент, т.е. RV, RALL и D, как связанный сбой, а решающий сбой является единичной точкой сбоя. Два других сечения имеют два члена, один для неудачных реализаций обеих версий, и один для обоих аппаратных узлов неудач. Вероятность того, что недопустимый результат будет получен в ходе одной итерации задачи, определяется выражением

$$P_{RV} + Q_{RV}P_D + Q_{RV}P_{RALL}Q_D + Q_{RV}Q_{RALL}Q_D P_{H^2} + P_{V^2}Q_{RV}Q_{RALL}Q_D\left(1-P_{H^2}\right) \qquad (18)$$

где P_X вероятность того, что произойдет событие X, и $Q_X = 1 - P_X$

3.1.2 Модель «дерева сбоев» для системы N-версионного программирования

Пример системы (N-version programming) NPV состоит из трех одинаковых аппаратных компонентов, каждая из которых имеет отдельную версию программного обеспечения. Это прямое отображение метода NPV на аппаратное обеспечение.

Если какой-либо из компонентов оборудования испытывает сбой, это приводит к тому, что версия программного обеспечения, запущенная на этом оборудовании, дает неточные результаты. Если все остальные аппаратные и программные модули функционируют должным образом, система все равно даст правильный результат, поскольку две из трех версий, большинство, являются правильными. Если у двух программных или аппаратных компонентов есть сбои, система выйдет из строя, так как будет только один правильный результат. Система также выйдет из строя, если по меньшей мере один компонент оборудования и один программный компонент выходят из

строя, но только если неисправное оборудование не содержит неисправную версию программного обеспечения.

На рисунке 22 показана модель дерева сбоев NVP. При наличии трех версий программного обеспечения, работающих на трех отдельных процессорах, необходимо учитывать несколько различных сценариев сбоя, включая совпадение несвязанных неисправностей, а также связанных ошибок программного обеспечения и комбинаций аппаратных и программных сбоев. Одна итерация задачи может не срабатывать по нескольким причинам: во-первых, если две из трех версий активируют несвязанные сбои, или если активирован какой-либо связанный с этой неисправностью сбой между двумя версиями; во-вторых, если вход активирует сбой, который влияет на все три версии или сбой в блоке принятия решения; в-третьих, если два из трех процессоров испытывают сбои в одной и той же задаче; наконец, если аппаратный узел выходит из строя, и один из программных компонентов на другом узле также выходит из строя (через несвязанную или связанную с ним ошибку).

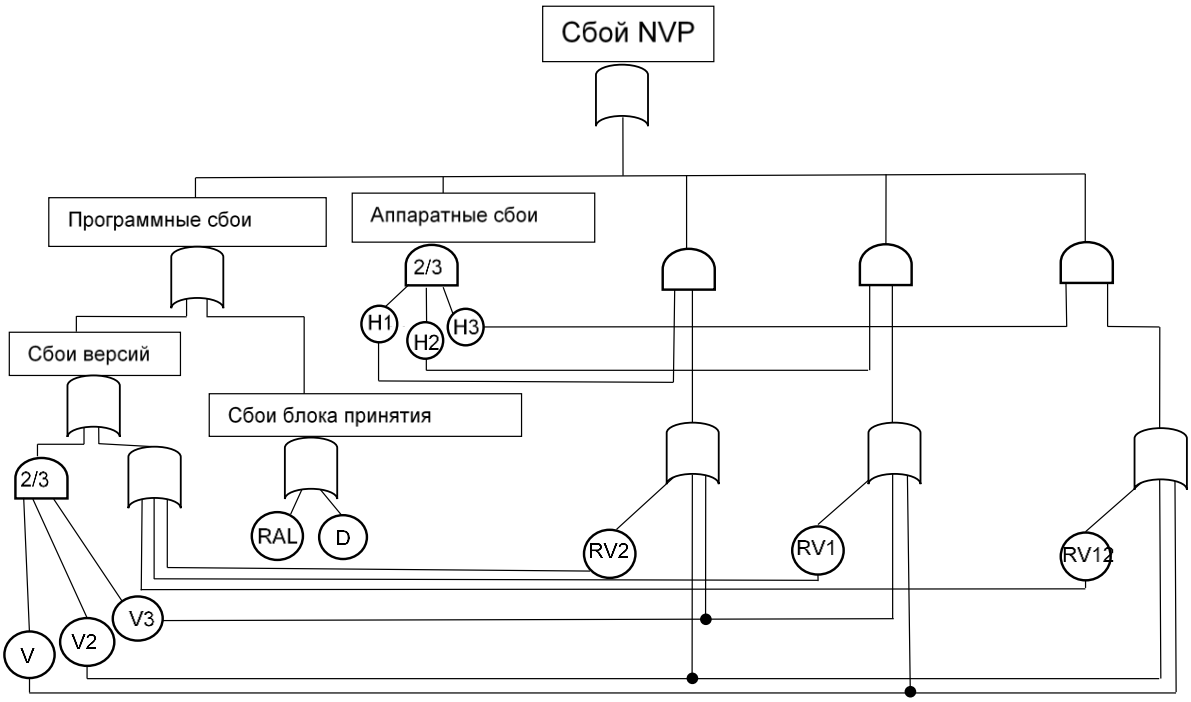


Рисунок 22 - Модель дерева сбоев для системы N-версионного программирования.

Дерево сбоев NVP имеет 20 минимальных сечений, из которых 5 имеют один элемент (RV12, RV13, RV23, RALL и D). Остальные сечения перечисляют 15 способов, с помощью которых программное обеспечение и / или аппаратные компоненты объединяются, чтобы вывести из строя две из трех версий. Решение модели дерева сбоев NVP представляется формулой

$$
\begin{aligned}
T_s = {} & P_{RV} + Q_{RV} P_{RV} + Q_{RV^2} P_{RV} + Q_{RV^3} P_D + Q_{RV^3} P_{RALL} Q_D + P_{V^2} Q_{RV^3} Q_{RALL} Q_D + \\
& + P_{V^2} Q_V Q_{RV^3} Q_{RALL} Q_D + + P_{V^2} Q_V Q_{RV^3} Q_{RALL} Q_D + Q_V P_V Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H \left(1 - P_V\right) + \\
& + Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H \left(1 - P_{V^2}\right) + Q_V P_V Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H \left(1 - P_{V^2}\right) + \\
& + Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H \left(1 - P_V\right)\left(1 - P_{V^2}\right) + Q_V P_V Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H \left(1 - P_V\right) + \\
& + Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H \left(1 - P_V\right)\left(1 - P_{V^2}\right) + Q_V P_V Q_V Q_{RV^3} Q_{RALL} Q_D P_H Q_H \left(1 - P_H\right) + \\
& + Q_{V^2} P_V Q_{RV^3} Q_{RALL} Q_D P_H Q_H \left(1 - P_H\right) + P_V Q_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H \left(1 - P_V\right) + \\
& + Q_{V^2} P_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H + P_V Q_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H \left(1 - P_V\right) + \\
& + Q_V P_V Q_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H .
\end{aligned}
\tag{19}
$$

3.1.3 Модель «дерева сбоев» для системы N версионного программирования с самопроверкой (NSCP)

Пример архитектуры NSCP состоит из четырех версий программного обеспечения и четырех аппаратных компонентов, каждая из которых сгруппирована в две пары, по существу разделяя систему на две половины.

Аппаратные пары работают в режиме избыточного горячего резерва, при этом каждый аппаратный компонент поддерживает одну версию программного обеспечения.

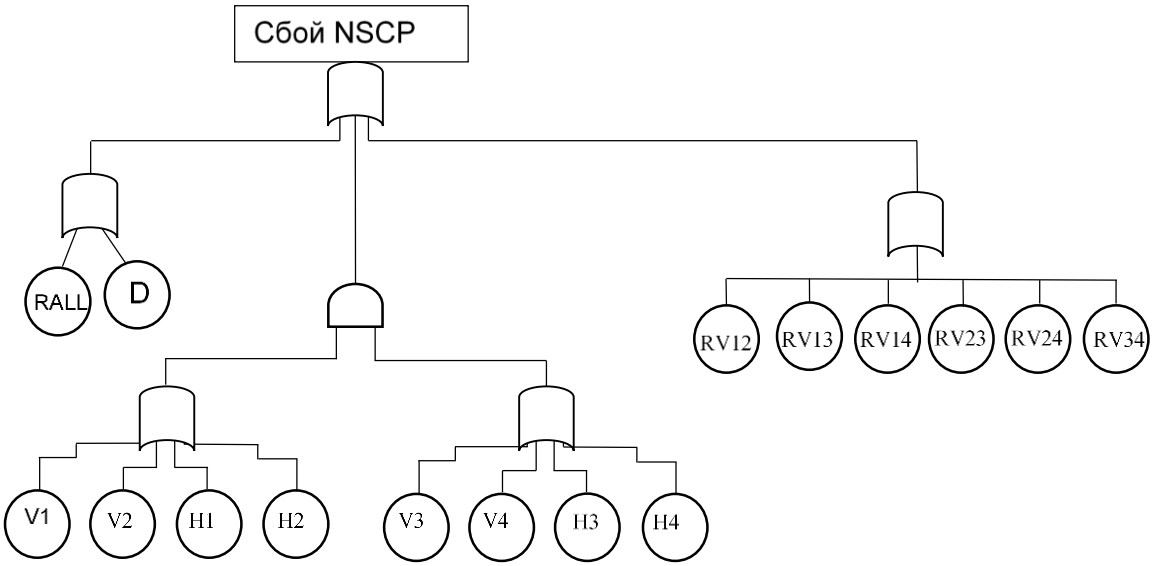


Рисунок 23 - Модель дерева сбоев для системы N версионного программирования с самопроверкой (NSCP)

Пары версий образуют самоконтролируемые программные компоненты. Программный компонент для самопроверки состоит из двух версий и алгоритма сравнения или версии и приемосдаточного теста. В этом случае обнаружение ошибок выполняется путем сравнения. Выполняются четыре версии программного обеспечения, а результаты V1 и V2 сравниваются друг с другом, как и результаты V3 и V4. Если обе пары результатов не совпадают, они отбрасываются, и используются только оставшиеся два. Если результаты совпадают, тогда результаты двух пар сравниваются. Ошибка оборудования приводит к тому, что версия программного обеспечения, запущенная на нем, дает неправильные результаты, как и ошибка в самой версии программного обеспечения. Это приводит к расхождению в результатах двух версий, в результате чего эта пара игнорируется.

Модель дерева сбоев системы NSCP (рисунок 23) показывает, что система уязвима для связанных ошибок, независимо от того, включают ли они версии в той же области ограничения ошибок или нет. Чтобы позволить последующее сравнение с системами NVP и DRB, мы проигнорировали возможность связанной ошибки, влияющей на три версии.

Для системы NSCP предусмотрено 24 сечения. Шесть одноэлементных сечений перечисляют связанные ошибки, влияющие на две версии, а два одиночных сечения отражают единственные точки сбоя в решающем устройстве и в связанной ошибке, влияющей на все версии (RALL). Имеется также 16 наборов сечений с двумя элементами, перечисляющих все комбинации ошибок версии и / или аппаратного обеспечения, влияющие на одну версию в каждой области ограничения ошибок. Решение модели дается формулой

$$
\begin{aligned}
T_s = {} & P_{RV} + Q_{RV} P_{RV} + Q_{RV^2} P_{RV} + Q_{RV^3} P_{RV} + Q_{RV^4} P_{RV} + Q_{RV^5} P_{RV} + Q_{RV^6} P_{RALL} + Q_{RV^6} P_D Q_{RALL} + \\
& + P_{V^2} Q_{RV^6} Q_D Q_{RALL} + P_{V^2} Q_V Q_{RV^6} Q_D Q_{RALL} + P_V Q_{V^2} Q_{RV^6} Q_D Q_{RALL} P_H + \\
& + P_V Q_{V^2} Q_{RV^6} Q_D Q_{RALL} Q_H P_H + Q_V P_{V^2} Q_{RV^6} Q_D Q_{RALL} + P_{V^2} Q_V Q_{RV^6} Q_D Q_{RALL} (1 - P_V) + \\
& + P_V Q_{V^2} Q_{RV^6} Q_D Q_{RALL} P_H (1 - P_V) + P_V Q_{V^2} Q_{RV^6} Q_D Q_{RALL} Q_H P_H (1 - P_V) + Q_{V^2} P_V Q_{RV^6} Q_D Q_{RALL} P_H + \\
& + Q_V P_V Q_{RV^6} Q_D Q_{RALL} P_H (1 - P_V)^2 + Q_{V^2} Q_{RV^6} Q_D Q_{RALL} P_{H^2} (1 - P_H)^2 + \\
& + Q_{V^2} Q_{RV^6} Q_D Q_{RALL} P_{H^2} Q_H (1 - P_V)^2 + Q_{V^2} P_V Q_{RV^6} Q_D Q_{RALL} Q_H P_H + \\
& + Q_V P_V Q_{RV^6} Q_D Q_{RALL} P_H (1 - P_V)^2 (1 - P_H) + Q_{V^2} Q_{RV^6} Q_D Q_{RALL} P_{H^2} (1 - P_V)^2 (1 - P_H) + \\
& + Q_{V^2} Q_{RV^6} Q_D Q_{RALL} P_{H^2} Q_H (1 - P_V)^2 (1 - P_H).
\end{aligned}
\tag{20}
$$

3.2 Модель «дерева сбоев» для системы с t/(n-1) алгоритмом принятия решения

Пример системы с t/(n-1) алгоритмом принятия решения состоит из трех одинаковых аппаратных компонентов, каждая из которых имеет отдельную версию программного обеспечения. Это прямое отображение метода t/(n-1) на аппаратное обеспечение.

Если какой-либо из компонентов оборудования испытывает сбой, это приводит к тому, что версия программного обеспечения, запущенная на этом оборудовании, дает неточные результаты. Если все остальные аппаратные и программные модули функционируют должным образом, система все равно даст правильный результат, поскольку две из трех версий, большинство, являются правильными.

Если у двух программных или аппаратных компонентов есть сбои, система выйдет из строя, так как будет только один правильный результат. Если выходы первой и второй версии совпадают, то на выход системы подается результат работы первой версии, если выходы не совпадают, то на выход системы подается результат работы третьей версии, результат работы второй версии никогда не подается на выход системы, а используется лишь для сравнения.

Программные сбои происходят в следующих случаях: когда ошибка происходит в первой или второй версии, компаратор возвращает ошибку, и система направляет на выход результат третьей версии, если и она дала сбой – происходит программный сбой; в случае, когда происходит межверсионная ошибка в 1 и 2 версии, компаратор принимает ответы как верные и на выход системы направляется результат работы первой, сбойной версии; в случае, когда все версии дают сбой. Система также выйдет из строя, если по меньшей мере один компонент оборудования и один программный компонент выходят из строя, но только если неисправное оборудование не содержит неисправную версию программного обеспечения при тех же комбинациях, что описаны выше.

На рисунке 24 показана модель дерева сбоев t/(n-1) алгоритма. При наличии трех версий программного обеспечения, работающих на трех отдельных процессорах, необходимо учитывать несколько различных сценариев сбоя, включая совпадение несвязанных неисправностей, а также связанных ошибок программного обеспечения и комбинаций аппаратных и программных сбоев.

Одна итерация задачи может не срабатывать по нескольким причинам: во-первых, если вход активирует сбой, который влияет на все три версии или сбой в блоке принятия решения; во-вторых, если два из трех процессоров испытывают сбои в одной и той же задаче; наконец, если аппаратный узел выходит из строя, и один из программных компонентов на другом узле также

выходит из строя (через несвязанную или связанную с ним ошибку) в тех же сочетаниях, что описаны выше. В случае аппаратного сбоя 2 из 3 процессоров всегда проходит аппаратный сбой в отличии от программного, поскольку в случае сбоя первой или второй и третьей версии пройдет сбой третьей версии, а в случае сбоя первой и второй версии произойдет сравнение двух отсутствующих из-за аппаратных сбоев ответов и компаратор вернет совпадение, поэтому на выход системы пойдет несуществующий результат сбойной первой версии.

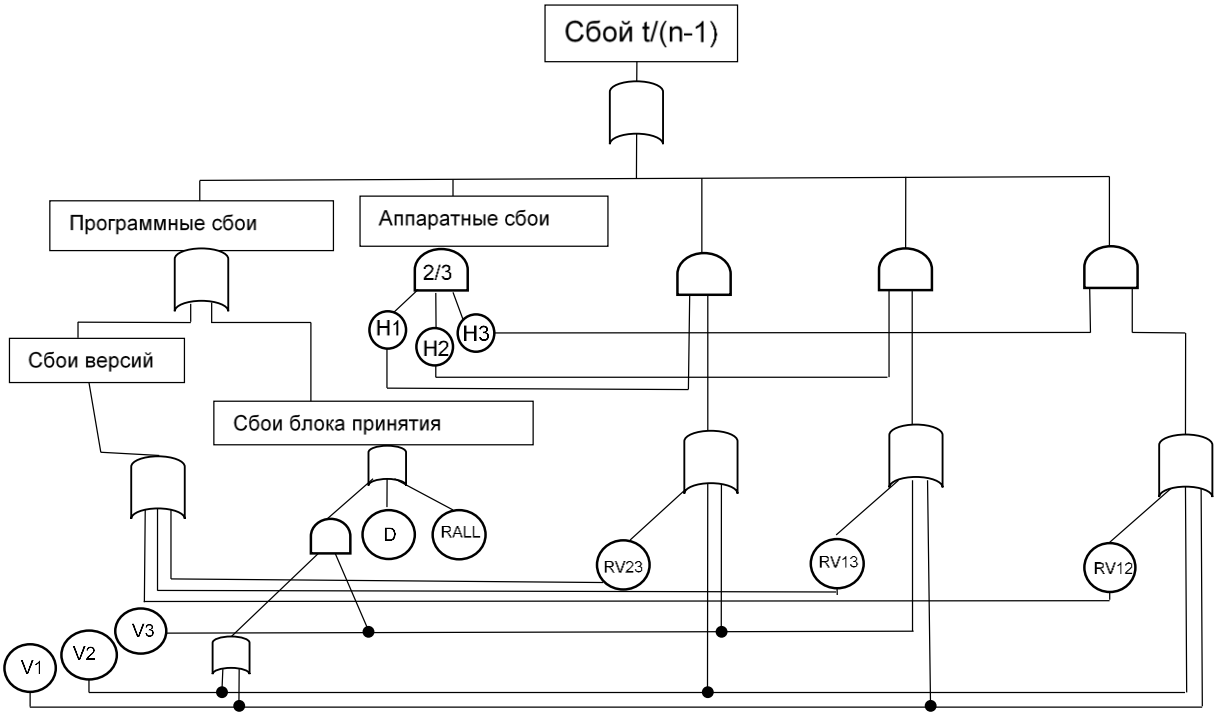


Рисунок 24 - Модель дерева сбоев для системы с t/(n-1) алгоритмом принятия решения.

Дерево сбоев t/(n-1) алгоритма имеет 20 минимальных сечений, из которых 5 имеют один элемент (RV12, RV13, RV23, RALL и D). Остальные сечения перечисляют 15 способов, с помощью которых программное обеспечение и / или аппаратные компоненты объединяются, чтобы привести к сочетанию, которое позволит пройти сбою. Решение модели дерева сбоев t/(n-1) алгоритма представляется формулой

$$
\begin{aligned}
T_s = {} & P_{RV} + Q_{RV} P_{RV} + Q_{RV^2} P_{RV} + Q_{RV^3} P_D + Q_{RV^3} P_{RALL} Q_D + P_{V1} P_{V2} Q_{RV12} Q_D P_{V3} + \\
& + P_{V1} Q_{V2} Q_D P_{V3} + Q_{V1} P_{V2} Q_D P_{V3} + Q_V P_V Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H + Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H + \\
& + Q_V P_V Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H + Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H (1 - P_V) + \\
& + Q_V P_V Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H + Q_{RV^3} Q_{RALL} Q_D P_{H^2} Q_H (1 - P_V) + \\
& + P_V Q_V Q_{RV^3} Q_{RALL} Q_D P_H Q_H (1 - P_H) + P_V Q_{RV^3} Q_{RALL} Q_D P_H Q_H (1 - P_H) + \\
& + P_V Q_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H + P_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H + \\
& + P_V Q_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H + P_V Q_V Q_{RV^3} Q_{RALL} Q_D Q_{H^2} P_H .
\end{aligned}
\tag{21}
$$

3.3 Программная реализация модели «деревьев сбоев»

Полученная методика реализована программно, разработанный инструмент моделирует систему с программной избыточностью, позволяет задавать все надежностные характеристики: вероятности сбоев каждой версии, вероятности межверсионных ошибок, вероятность ошибки блока принятия решения, вероятность аппаратных сбоев, количество версий для NVP, вероятность возникновения связанной ошибки, влияющей на все версии, а также на решение, вызванное несовершенными спецификациями, а так-же минимальную надежность системы, которую необходимо достигнуть при обратном расчете. Система позволяет изучить поведение системы (изменение вероятности прохождения сбоя) при изменениях входных параметров. Рассмотрим интерфейс программы и полученные результаты на рисунках 25-28:

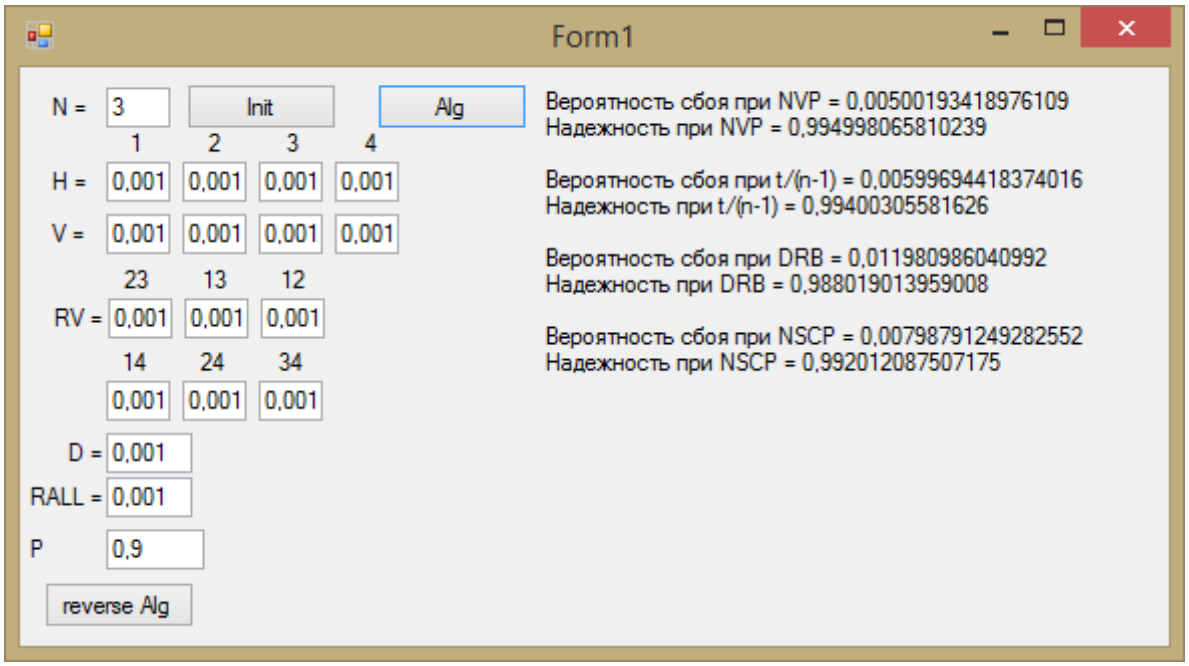


Рисунок 25 - Прямой расчет деревьев сбоев при вероятности сбоев версий Pvx= 0.001

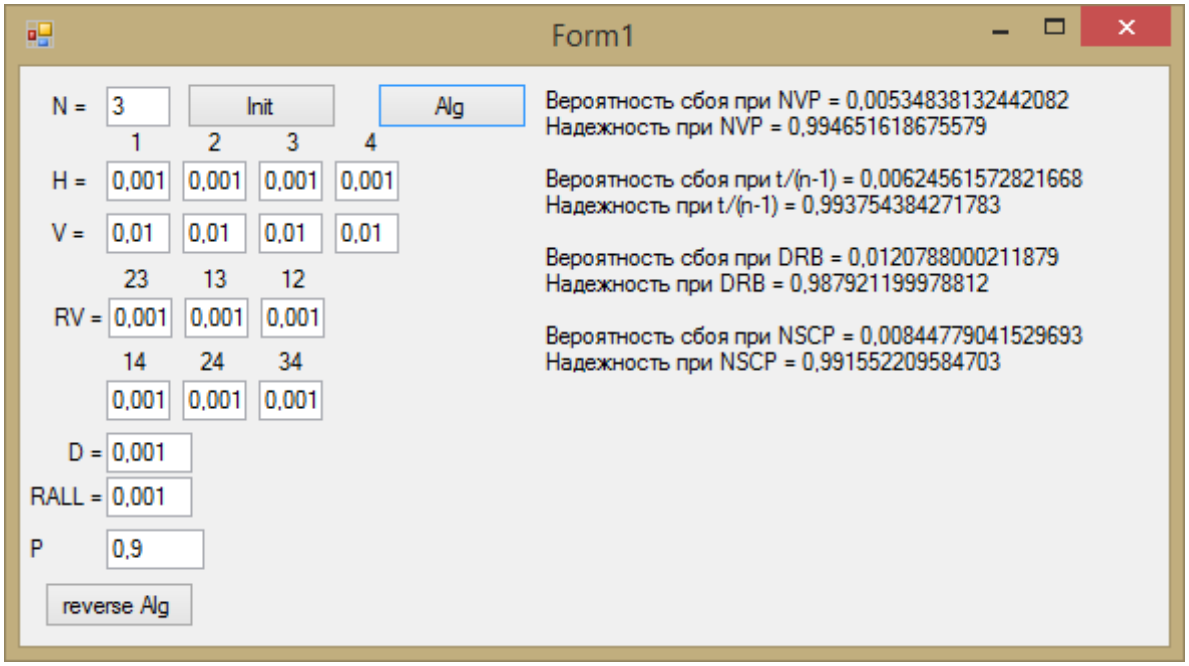


Рисунок 26 - Прямой расчет деревьев сбоев при вероятности сбоев версий Pvx= 0.01

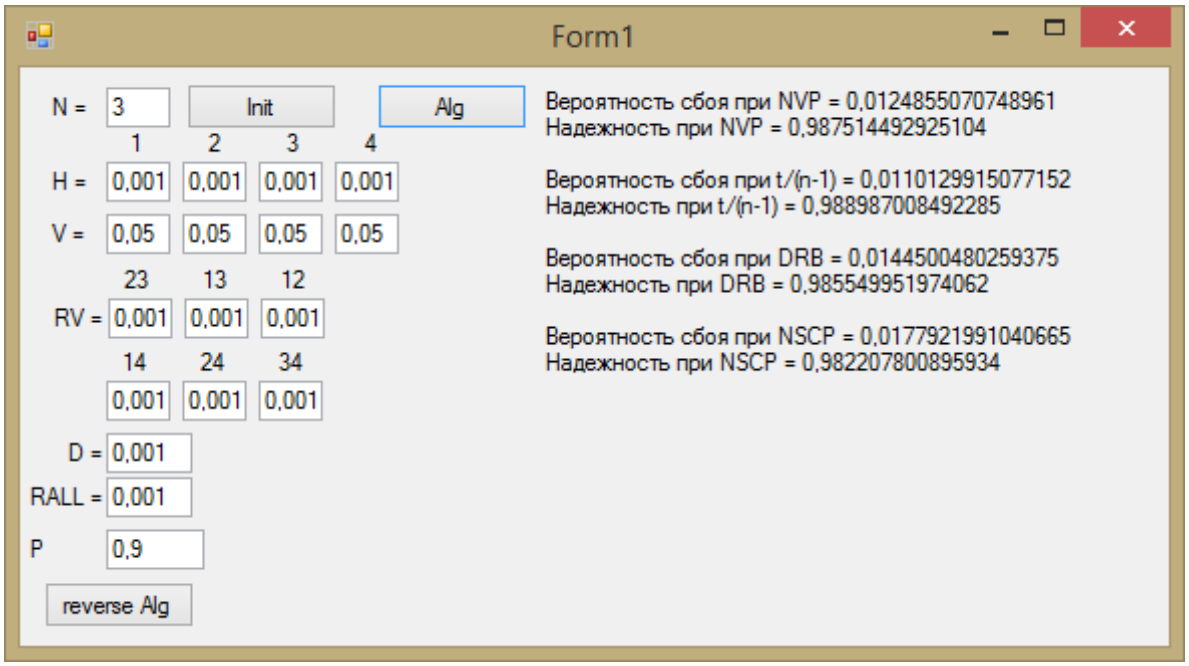


Рисунок 27 - Прямой расчет деревьев сбоев при вероятности сбоев версий Pvx= 0.05

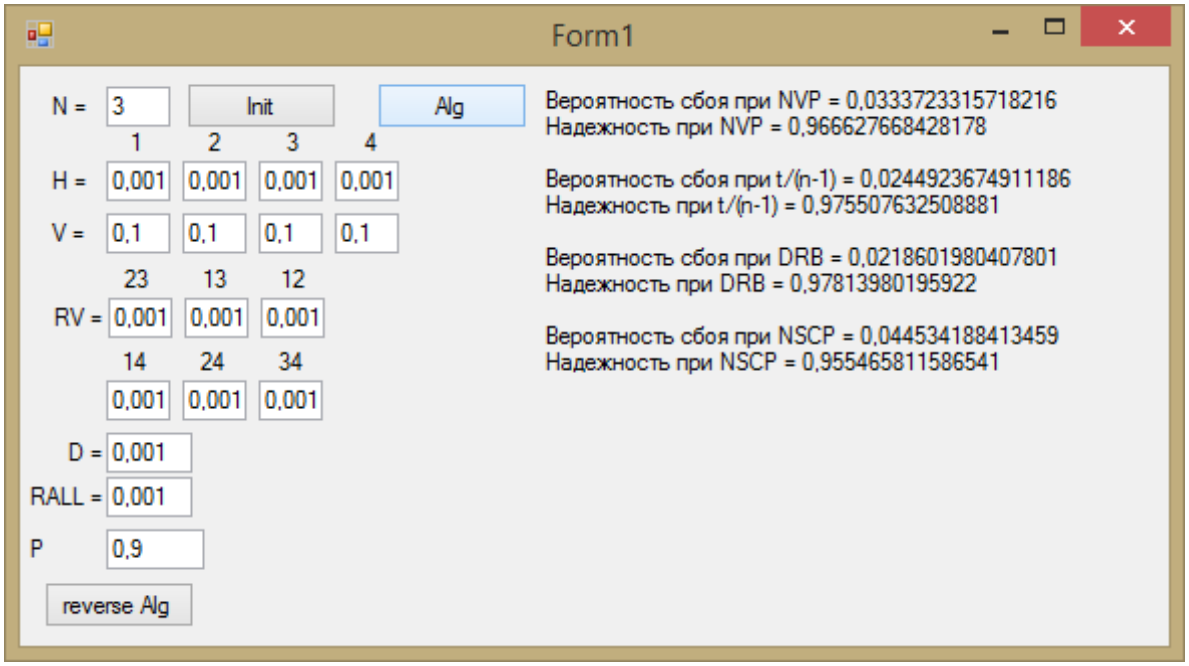


Рисунок 28 - Прямой расчет деревьев сбоев при вероятности сбоев версий Pvx= 0.1

Проанализировав результата, представленные на рисунках 25-28, можно сделать вывод о том, что изменение вероятности возникновения одиночных сбоев в версиях на порядок (с 0.001 до 0.01) не показывает существенного влияния на надежность системы в целом, изменение надежности системы

менее 10%, однако дальнейшее увеличение вероятности сбоев версий более существенно снижает надежность системы, к примеру, повышение данного параметра с 0.05 до 0.1 (в 2 раза) приводит к снижению надежности системы в более чем 2.5 раза. Рассмотрим чувствительность системы к изменению вероятностей аппаратных сбоев:

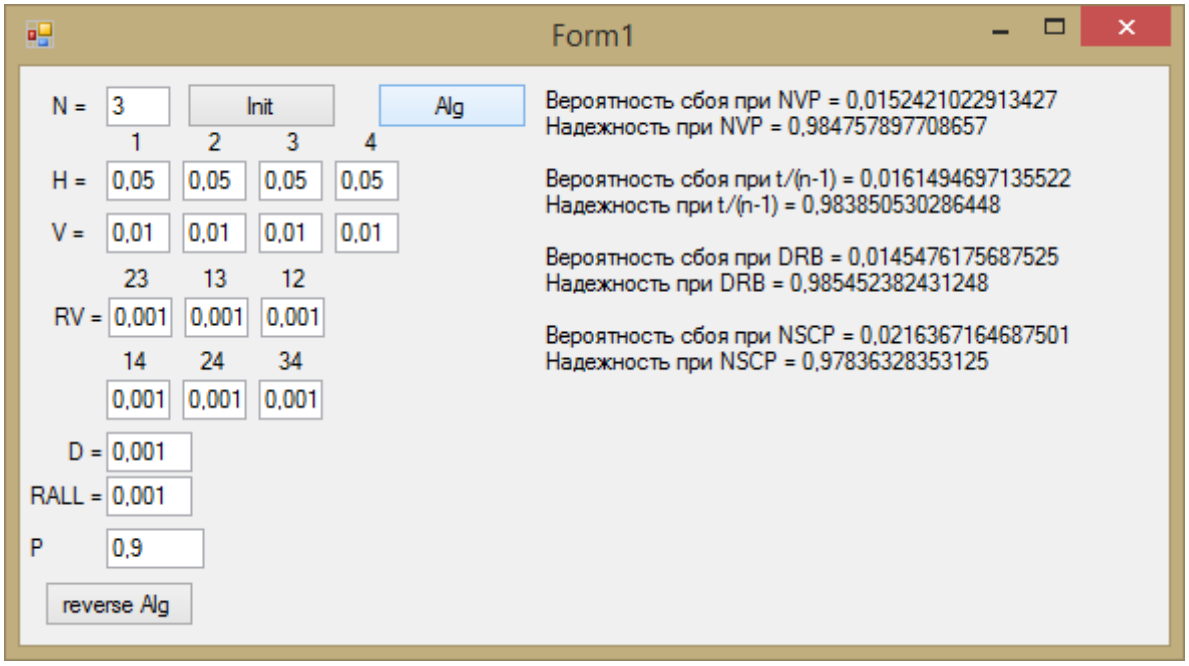


Рисунок 29 - Прямой расчет деревьев сбоев при вероятности аппаратных сбоев Phx= 0.05

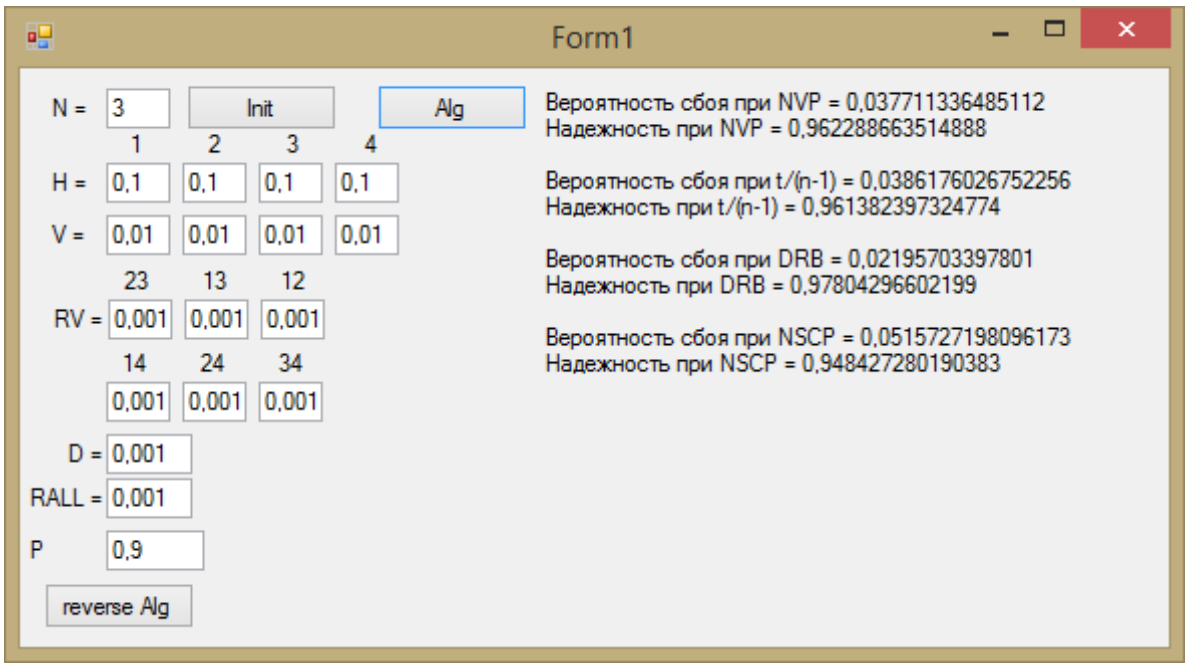


Рисунок 30 - Прямой расчет деревьев сбоев при вероятности аппаратных сбоев Phx= 0.1

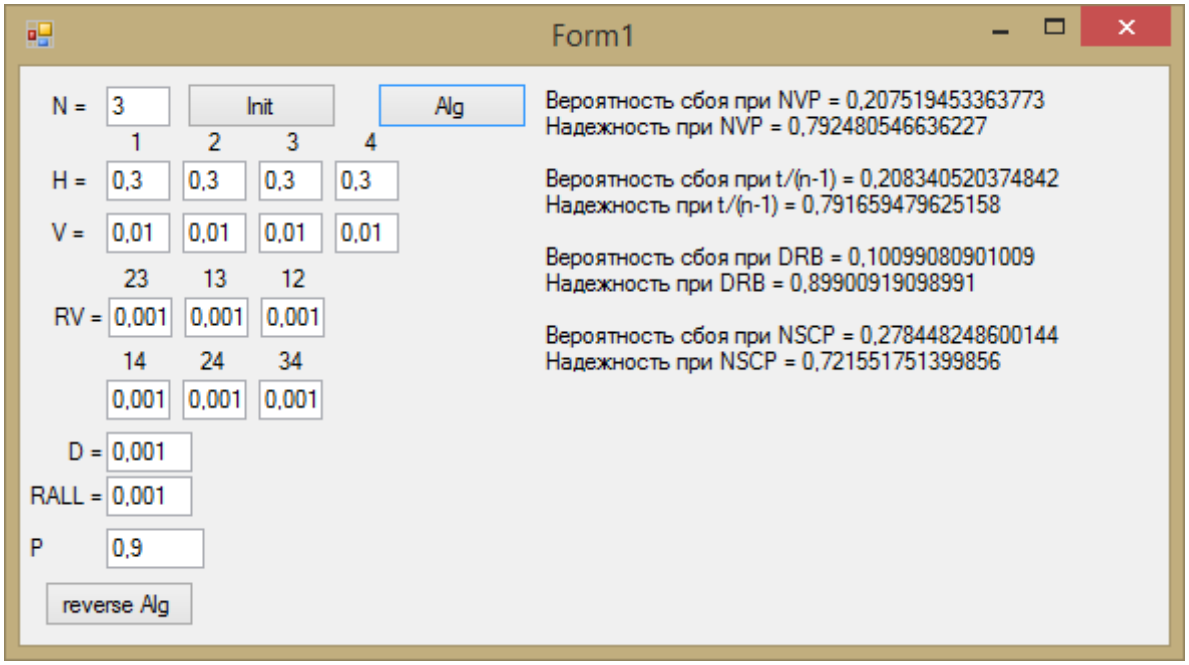


Рисунок 31 - Прямой расчет деревьев сбоев при вероятности аппаратных сбоев Phx= 0.3

Результаты, представленные на рисунках 29-31 показывают, что система так же чувствительна к вероятности возникновения аппаратных ошибок, превышающей значение 0.05.

Рассмотрим чувствительность системы к изменению вероятностей возникновения связанной ошибки, влияющей на все версии, а также на решение, вызванное несовершенными спецификациями:

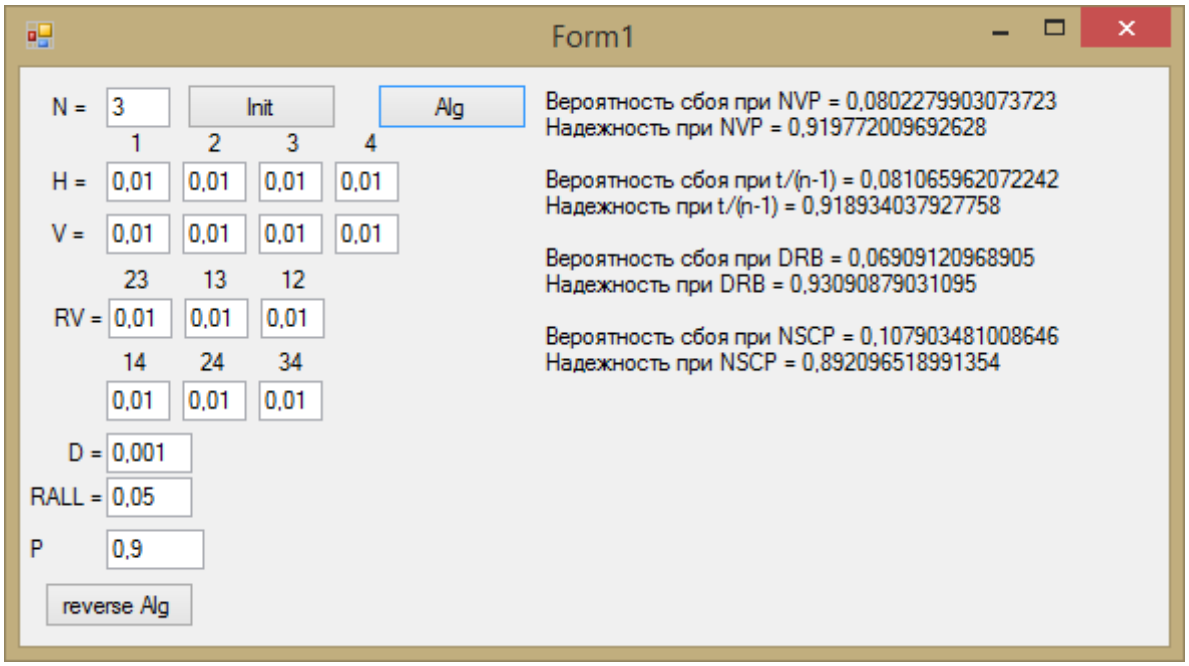


Рисунок 32 - Прямой расчет деревьев сбоев при вероятности сбоя Pall= 0.05

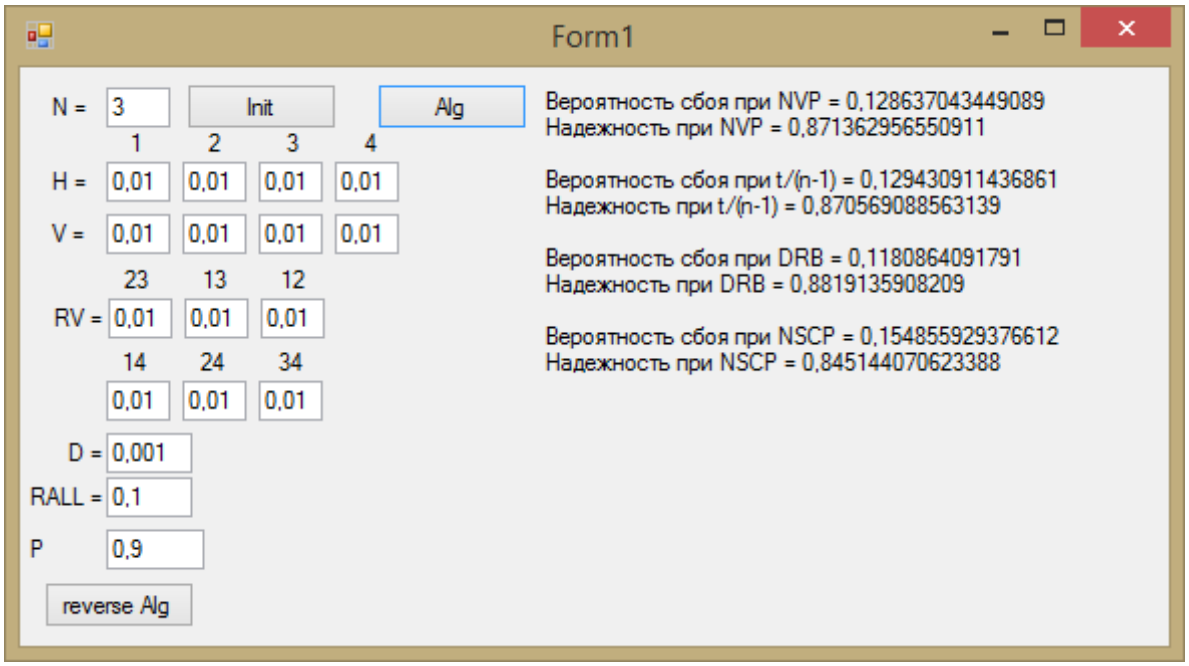


Рисунок 33 - Прямой расчет деревьев сбоев при вероятности сбоя Pall= 0.1

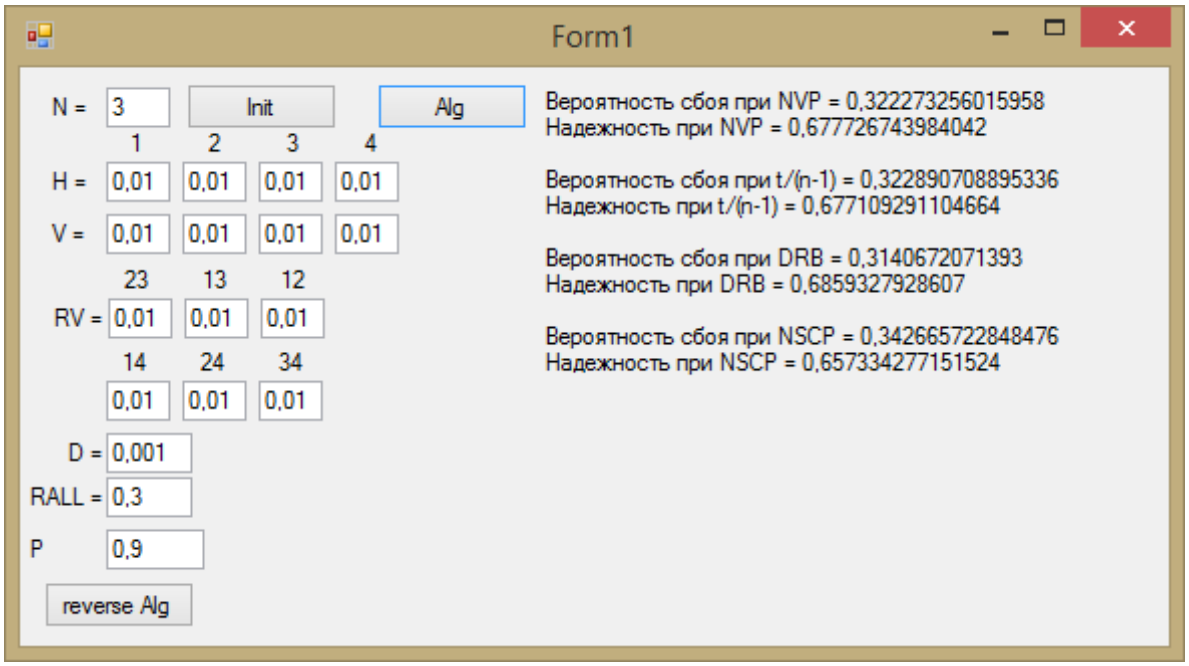


Рисунок 34 - Прямой расчет деревьев сбоев при вероятности сбоя Pall= 0.3

Результаты, представленные на рисунках 32-34 показывают, что система еще более чувствительна к вероятности возникновения связанной ошибки, влияющей на все версии, а также на решение, вызванное несовершенными спецификациями, поскольку она влияет на блок принятия решений и сбой от нее проходит вне зависимости от сбоев отдельных программных и аппаратных модулей.

Рассмотрим пример работы обратного расчета деревьев сбоев, когда мы фиксируем все характеристики системы, кроме искомых, в нашем случае это вероятности одиночного сбоя версий, для простоты расчета примем их равными ($P_{V1} = P_{V2} = P_{V3} = P_{V4}$), поскольку в противном случае мы получим не конкретный ответ, а бесконечное количество сочетаний вероятностей.

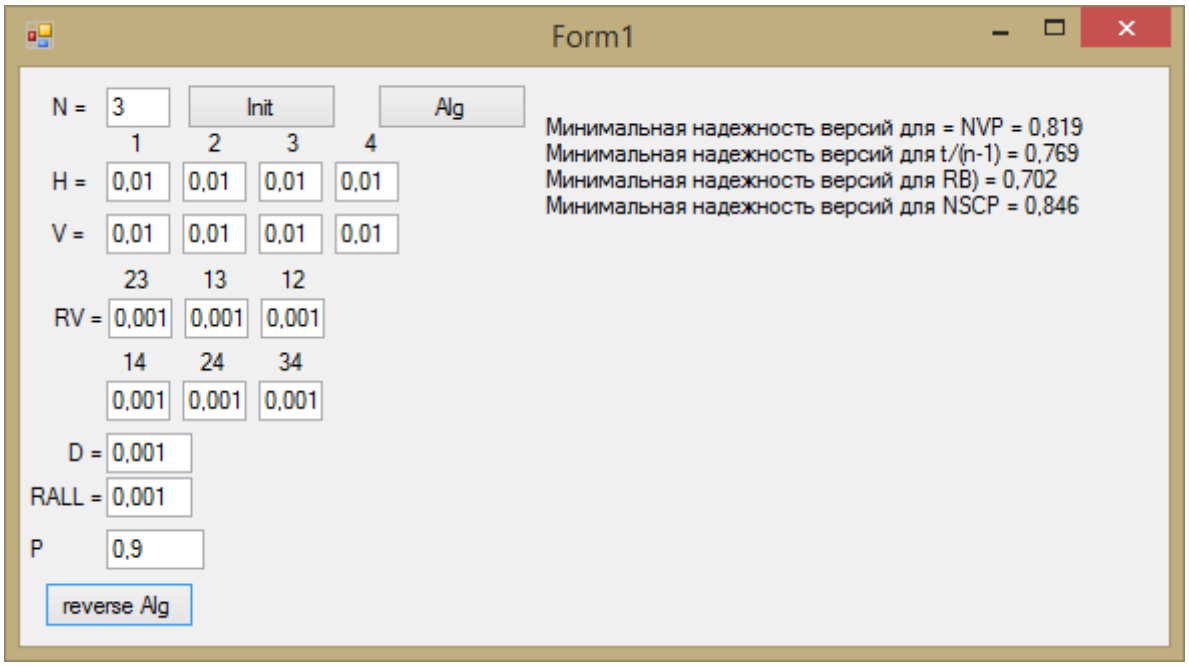


Рисунок 35 - Обратный расчет деревьев сбоев при минимальном требовании надежности системы P=0.9

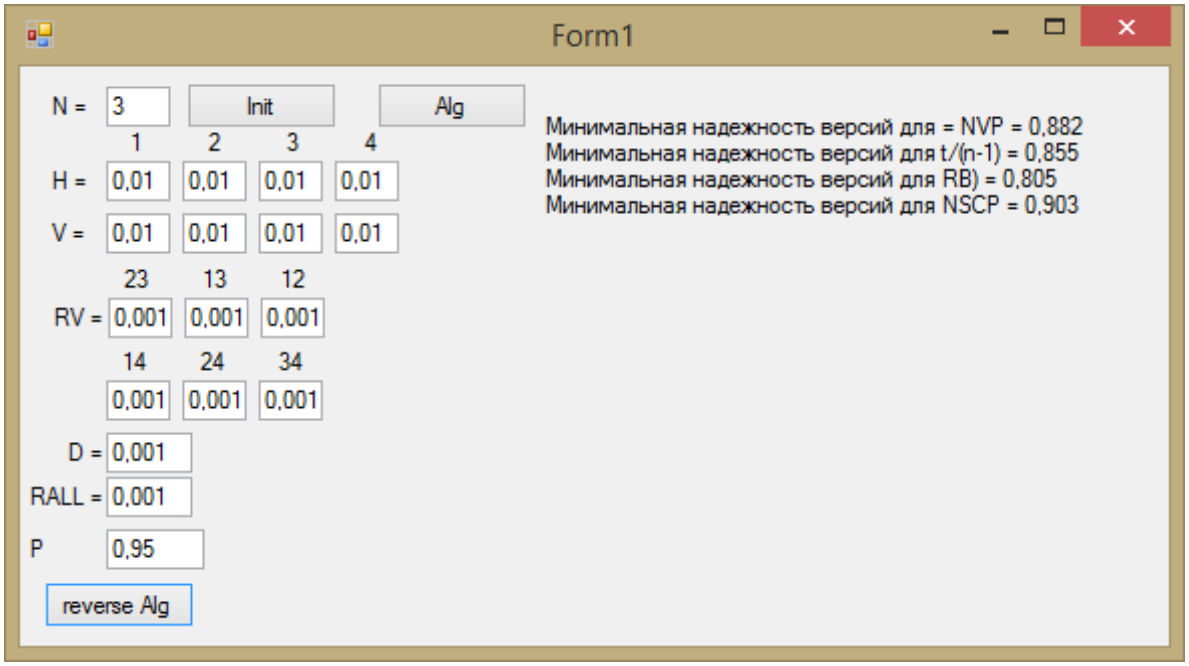


Рисунок 36 - Обратный расчет деревьев сбоев при минимальном требовании надежности системы P=0.95.

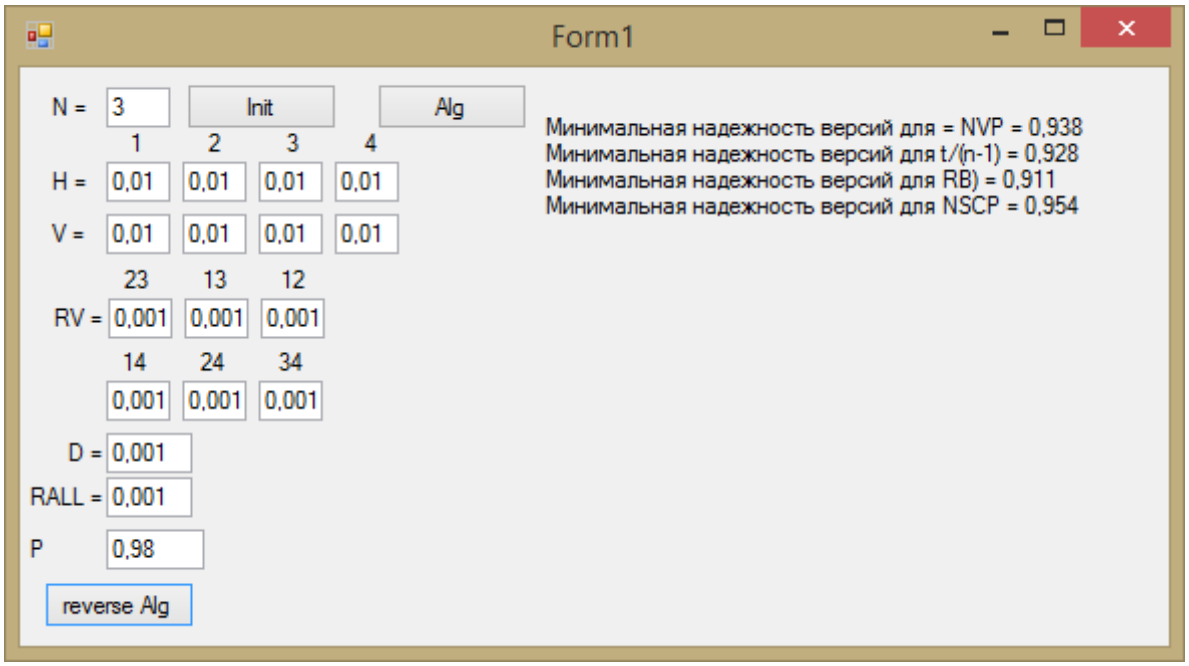


Рисунок 37 - Обратный расчет деревьев сбоев при минимальном требовании надежности системы P=0.98

Из результатов, представленных на рисунках 35 - 37 можно сделать вывод о том, что из ненадежных программных модулей получается гораздо более надежная система, к примеру, для обеспечения надежности системы в 0.9, при применении модели восстанавливающихся блоков достаточны программные версии с надежностью 0.702, что подтверждает эффективность подхода с использованием программной избыточности.

3.4 Модель корректности

Оригинальность данной модели в механизме определения зависимости между величиной «корректности» - Sn и показателями, влияющими на надежность системы: количеством мультиверсий, надежностью каждой из версий, надежностью аппаратной платформы и блока принятия решения. Данный подход позволяет оценить эффективность основных моделей мультиверсионной методологии при различном численном соотношении мультиверсий, а также других параметров надежности.

Корректность - это сумма вероятности всех сочетаний возможных выходов мультиверсий, успешной работы аппаратного обеспечения и

вариантов работы блока принятия решения, которые дадут корректный выход. Некорректный выход системы может привести к возникновению сбоя, но если он будет обработан до возникновения отказа, система останется надежной [105]. Понятие корректности является статическим, а надежность - динамическим и более широким, поскольку описывает поведение системы во времени и реакцию на исключительные ситуации, имеющие место при возникновении на входе данных, не попадающих в диапазон, описанный в спецификации системы.

Для разных мультиверсионных моделей сочетание (2.1) будет различным, поскольку для модели восстанавливающихся блоков необходим только один корректный выход, для мультиверсионной модели с блоком голосования абсолютным большинством необходимо минимум ($\lfloor N/2 \rfloor+1$) корректных выходов версий [106] и нужно учесть все возможные сочетания, удовлетворяющие нашему условию.

Расчет корректности системы может показать нам как разницу между различными мультиверсионными моделями и количеством версий в рамках каждой из них, так и разницу в различных алгоритмах, заложенных в блоке принятия решения [107]. В нашем случае видно, что взвешенные алгоритмы голосования повышают корректность системы, поскольку повышается как компонента (2.2), благодаря повышению надежности блока голосования при идентичных выходах версий, так и (2.1), поскольку увеличивается количество сочетаний, достаточных блоку голосования для принятия верного решения.

Показатель корректности не совпадает с надежностью системы, а лишь показывает сумму всех сочетаний возможных вариантов работы всех компонентов системы, которые приведут к формированию корректного выхода. Этот показатель будет несколько ниже надежности системы в целом, поскольку оставшиеся сочетания, которые не гарантируют корректность выхода, с определенной вероятностью также дадут корректный выход, что будет учтено при расчете именно надежности системы.

Общую формулу корректности можно выразить следующим образом:

$$S_n = D_X^+ \cdot H_{D_X}^+ \sum_{i=m}^{n} P_i^+ \cdot Q_1^+ , \quad (22)$$

где – P_i^+ - вероятность сочетания корректных выходов версий, достаточного для правильной работы блока принятия решений, (2.1);

D_X^+ - вероятность корректного выхода блока принятия решения при X входах (2.2);

$H_{D_X}^+$ - вероятность корректной работы аппаратной платформы, на которой исполняется программа блока принятия решения (2.3);

$Q_i^+ = (1 - P_{i-1}^+)$, (2.4).

Для исследования зависимости величины корректности от основных вероятностных параметров надежности и применяемой схемой повышения отказоустойчивости реализуем программную среду, рассчитывающую корректность. Для этого выразим расчетные формулы для каждой из предложенных схем. Для N-версионного программирования при N=3:

$$\begin{aligned} S_{NVP(3)} &= D_{NVP(3)}^+ \cdot H_{D_3}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} + D_{NVP(3)}^+ \cdot H_{D_3}^+ \cdot P_{V1} \cdot P_{V2} \cdot (1 - P_{V3}) + \\ &+ D_{NVP(3)}^+ \cdot H_{D_3}^+ \cdot P_{V1} \cdot (1 - P_{V2}) \cdot P_{V3} + D_{NVP(3)}^+ \cdot H_{D_3}^+ \cdot (1 - P_{V1}) \cdot P_{V2} \cdot P_{V3}. \end{aligned} \quad (23)$$

Для N-версионного программирования при N=4:

$$\begin{aligned} S_{NVP(4)} &= D_{NVP(4)}^+ \cdot H_{D_4}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} + D_{NVP(4)}^+ \cdot H_{D_4}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot (1 - P_{V4}) + \\ &+ D_{NVP(4)}^+ \cdot H_{D_4}^+ \cdot P_{V1} \cdot P_{V2} \cdot (1 - P_{V3}) \cdot P_{V4} + D_{NVP(4)}^+ \cdot H_{D_4}^+ \cdot P_{V1} \cdot (1 - P_{V2}) \cdot P_{V3} \cdot P_{V4} + \\ &+ D_{NVP(4)}^+ \cdot H_{D_4}^+ \cdot (1 - P_{V1}) \cdot P_{V2} \cdot P_{V3} \cdot P_{V4}. \end{aligned} \quad (24)$$

Для N-версионного программирования при N=5:

$$\begin{aligned} S_{NVP(5)} &= D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} \cdot P_{V5} + D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} \cdot (1 - P_{V5}) + \\ &+ D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot (1 - P_{V4}) \cdot P_{V5} + D_{NVP(5)}^+ \cdot H_{D_5}^+ \cdot P_{V1} \cdot P_{V2} \cdot (1 - P_{V3}) \cdot P_{V4} \cdot P_{V5} + \\ &+ D_{NVP(5)}^+ \cdot H_{D_5}^+ \cdot P_{V1} \cdot (1 - P_{V2}) \cdot P_{V3} \cdot P_{V4} \cdot P_{V5} + D_{NVP(5)}^+ \cdot H_{D_5}^+ \cdot (1 - P_{V1}) \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} \cdot P_{V5} + \\ &+ D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot (1 - P_{V4}) \cdot (1 - P_{V5}) + D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot P_{V2} \cdot (1 - P_{V3}) \cdot P_{V4} \cdot (1 - P_{V5}) + \\ &+ D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot (1 - P_{V2}) \cdot P_{V3} \cdot P_{V4} \cdot (1 - P_{V5}) + D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot (1 - P_{V1}) \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} \cdot (1 - P_{V5}) + \\ &+ D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot P_{V2} \cdot (1 - P_{V3}) \cdot (1 - P_{V4}) \cdot P_{V5} + D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot P_{V1} \cdot (1 - P_{V2}) \cdot P_{V3} \cdot (1 - P_{V4}) \cdot P_{V5} + \\ &+ D_{NVP(5)}^+ \cdot H_{D5}^+ \cdot (1 - P_{V1}) \cdot P_{V2} \cdot P_{V3} \cdot (1 - P_{V4}) \cdot P_{V5} + D_{NVP(5)}^+ \cdot H_{D_5}^+ \cdot P_{V1} \cdot (1 - P_{V2}) \cdot (1 - P_{V3}) \cdot P_{V4} \cdot P_{V5} + \\ &+ D_{NVP(5)}^+ \cdot H_{D_5}^+ \cdot (1 - P_{V1}) \cdot P_{V2} \cdot (1 - P_{V3}) \cdot P_{V4} \cdot P_{V5} + D_{NVP(5)}^+ \cdot H_{D_5}^+ \cdot (1 - P_{V1}) \cdot (1 - P_{V2}) \cdot P_{V3} \cdot P_{V4} \cdot P_{V5}. \end{aligned} \quad (25)$$

Представленный расчет сечений аналитически подтверждает, что при увеличении числа версий корректность будет повышаться, поскольку увеличивается количество сочетаний, дающий корректный ответ системы, так как для успешного голосования абсолютным большинством (для простоты используем классический алгоритм голосования) достаточно (N/2)+1 корректно сработавших версий, то для корректного голосования подходят случаи как 5 корректно сработавших версий, 4 из 5, и даже любые 3 из 5.

Для t/(n-1) алгоритма при N=3:

$$\begin{aligned} S_{t/(n-1)} &= D^{+}_{t/(n-1)} \cdot H^{+}_{D_3} \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} + D^{+}_{t/(n-1)} \cdot H^{+}_{D_3} \cdot P_{V1} \cdot P_{V2} \cdot (1-P_{V3}) + \\ &+ D^{+}_{t/(n-1)} \cdot H^{+}_{D_3} \cdot P_{V1} \cdot (1-P_{V2}) \cdot P_{V3} + D^{+}_{t/(n-1)} \cdot H^{+}_{D_3} \cdot (1-P_{V1}) \cdot P_{V2} \cdot P_{V3} + \\ &+ D^{+}_{t/(n-1)} \cdot H^{+}_{D_3} \cdot (1-P_{V1}) \cdot (1-P_{V2}) \cdot P_{V3} \cdot P_{rv12}. \end{aligned} \quad (26)$$

В данной формуле последний компонент описывает ситуацию, когда первые две версии дают отличающиеся неверные ответы (ошибки именно независимые, не проявление межверсионной ошибки первой и второй версий), компаратор возвращает несовпадение и t/(n-1) алгоритм отправляет на выход результат 3 версии, если она сработала корректно, то система даст корректный ответ.

Для модели восстанавливающихся блоков:

$$S_{RB} = D^{+}_{RB} \cdot H^{+}_{D_2} \cdot P_{V1} + D^{+}_{RB} \cdot H^{+}_{D_2} \cdot (1-P_{V1}) \cdot P_{V2}. \quad (27)$$

Для данной модели мы имеем наиболее простую расчетную формулу, поскольку корректный ответ проходит только в двух случаях – корректной основной версии и корректной второй версии при неисправной первой и правильной работе блока принятия решения. Может показаться, что необходимо рассматривать случаи корректности обеих версий и случай корректности первой при некорректности второй, однако математически это и есть вероятность корректной работы первой версии, отдельное рассмотрение этих случаев усложнит расчетную формулу, не изменив результат.

Для NSCP при N=4:

$$
\begin{aligned}
S_{NSCP(4)} &= D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} + D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot P_{V1} \cdot P_{V2} \cdot P_{V3} \cdot (1-P_{V4}) + \\
&+ D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot P_{V1} \cdot P_{V2} \cdot (1-P_{V3}) \cdot P_{V4} + D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot P_{V1} \cdot (1-P_{V2}) \cdot P_{V3} \cdot P_{V4} + \\
&+ D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot (1-P_{V1}) \cdot P_{V2} \cdot P_{V3} \cdot P_{V4} + D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot (1-P_{V1}) \cdot (1-P_{V2}) \cdot P_{V3} \cdot P_{V4} \cdot P_{RV12} + \\
&+ D^{+}_{NSCP(4)} \cdot H^{+}_{D_4} \cdot P_{V1} \cdot P_{V2} \cdot (1-P_{V3}) \cdot (1-P_{V4}) \cdot P_{RV34}.
\end{aligned}
\tag{28}
$$

В данной формуле последние 2 элемента описывают ситуации, когда или первая и вторая, или третья и четвертая версии дали различные ошибочные ответы, но не проявилась межверсионная ошибка, тогда система сможет дать корректный ответ, в отличии от N-версионного голосования абсолютным большинством, где при 2 правильных ответах из четырех версий система не сможет принять решение.

3.5 Программная реализация модели корректности

Рассмотрим интерфейс программной реализации, представленный на рисунке 38, видны области для ввода параметров системы: количество версий, надежность аппаратных компонент, надежность версий, надежность блоков принятия решений для различных схем, вероятности межверсионной ошибки 1 и 2, 3 и 4 версий (точнее, ее обратную величину: 1-P, для удобства расчета), область вывода результатов расчета корректности для все обсчитываемых схем, область управления масштабом отрисовки графика по оси Y, кнопки инициализации значений переменных из областей ввода на форме, расчета корректности и построения графиков.

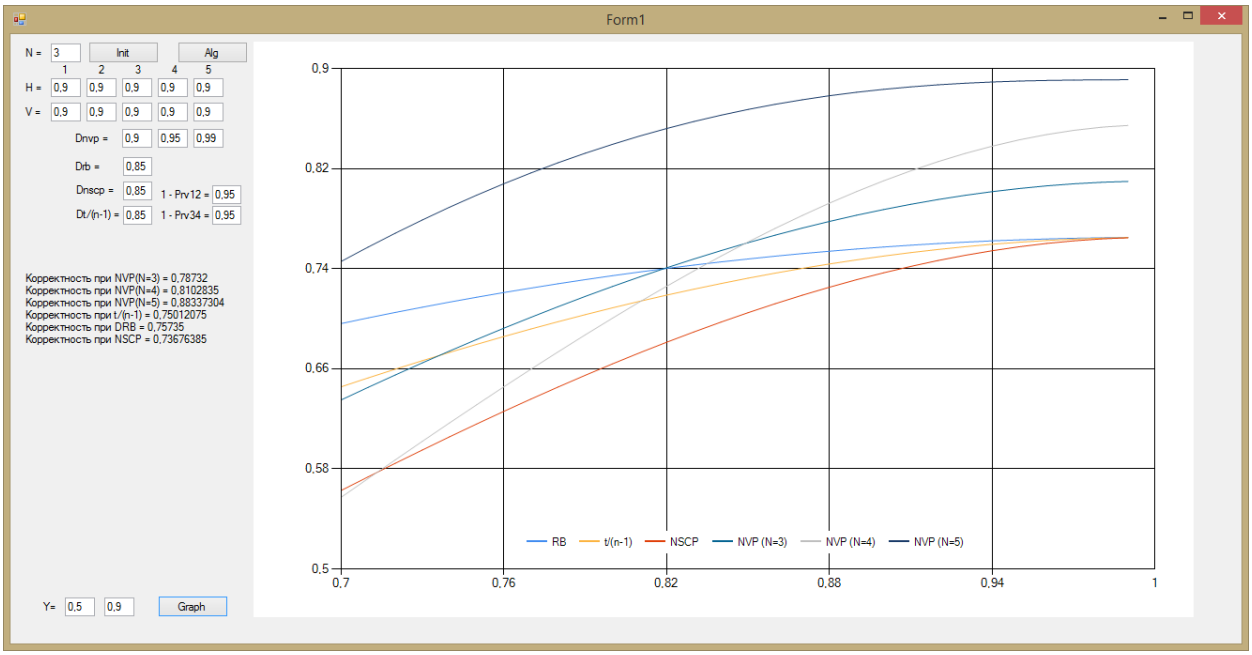


Рисунок 38 - Интерфейс программы, реализующей расчет корректности.

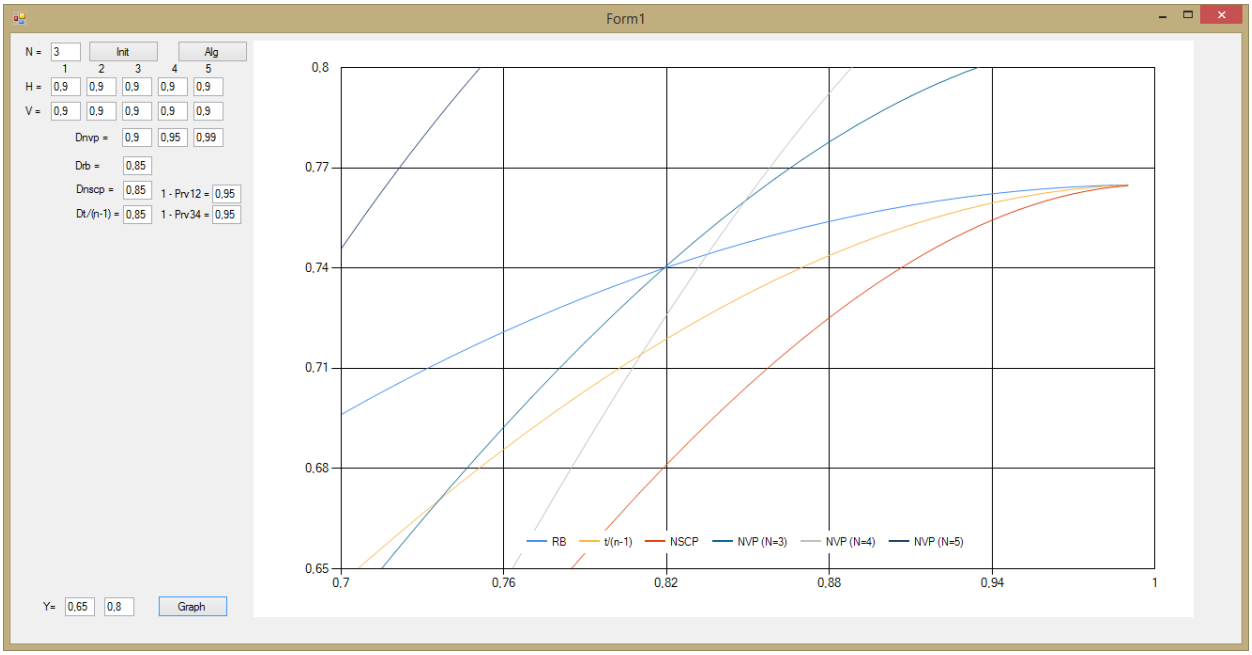


Рисунок 39 - Изменение масштаба отрисовки графиков по оси Y

Программа позволяет изменять масштаб отрисовки графиков по оси Y, это сделано для более наглядного отображения графиков или более детального просмотра наиболее интересной области, к примеру, сравним рисунки 38 и 39, на первом виден общий характер графиков, а на втором крупнее рассмотрены места их пересечений.

При нажатии кнопки «Alg» происходит расчет корректности для всех схем с применением всех характеристик системы, введенных на форме, при построении графиков не используются параметры надежностей версий (V1-V5), вместо них последовательно подставляются совпадающие для всех пяти версий значения от 0.7 до 0.99 с шагом 0.01, поскольку именно этот диапазон значений является наиболее интересным для изучения, так как версии с параметром надежности менее 0.7 нецелесообразно применять в отказоустойчивых системах, а при версии с надежностью 1, пропадает смысл в программной избыточности, поскольку уже имеется абсолютно надежная версия.

Рассмотрим влияние на поведение системы изменения некоторых входных параметров, для начала – изменим аппаратную надежность, сначала уменьшим до 0.8, потом увеличим до 0.99.

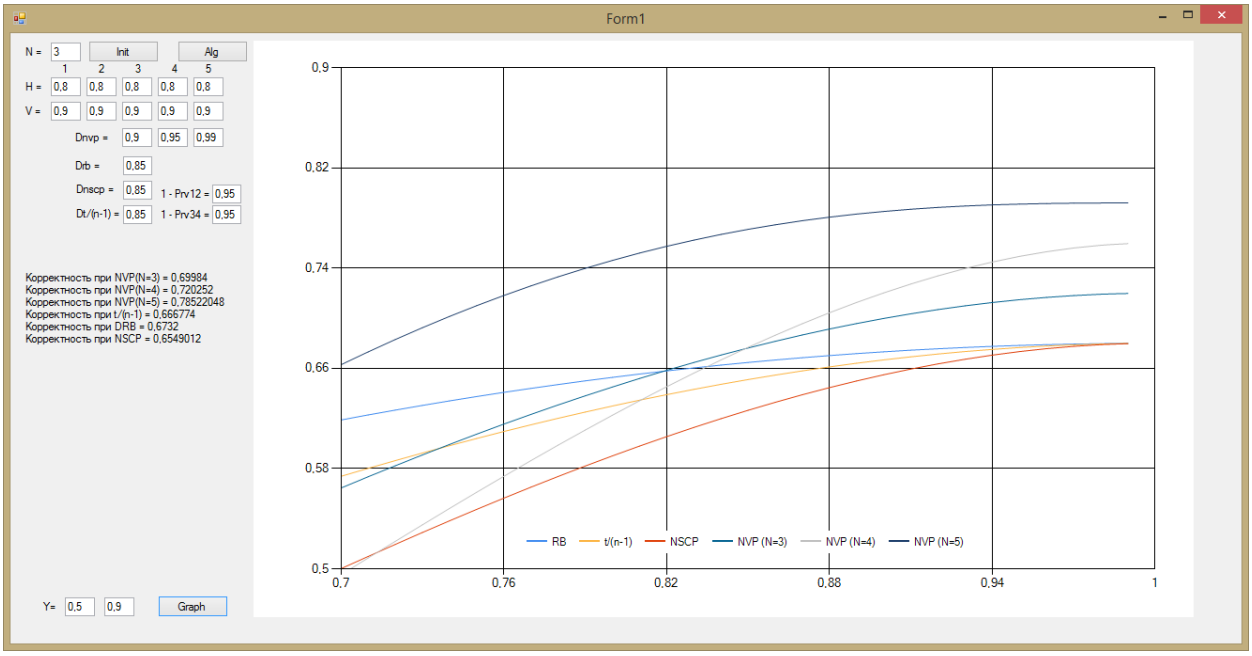


Рисунок 40 - Результат работы программы при аппаратной надежности 0.8

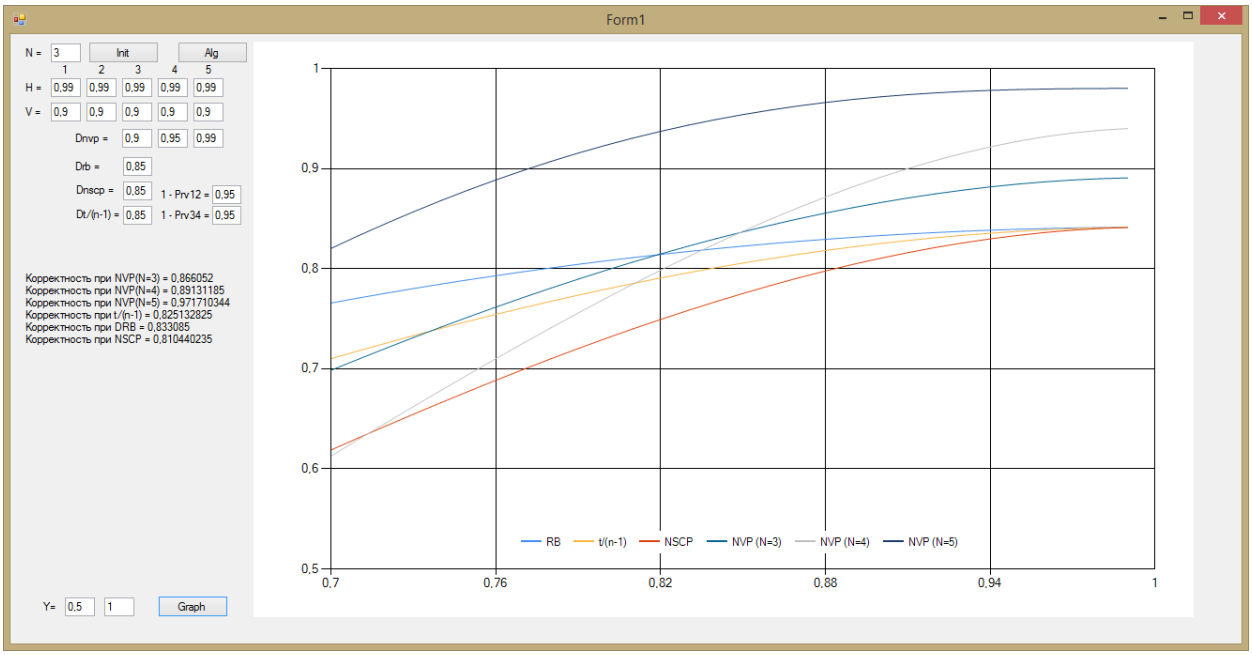


Рисунок 41 - Результат работы программы при аппаратной надежности 0.99

Как видно из рисунков 40 и 41, параметр аппаратной надежности оказывает существенное влияние на корректность системы, в случае повышения аппаратной надежности до 0.99 пришлось даже изменить масштаб отрисовки графиков, поскольку два из них превысили значение 0.9.

Интересным является случай, когда имеется оборудование с разной надежностью, применение различного оборудование обусловлено обеспечением разнообразия, для минимизации межверсионных сбоев, исследуем данную ситуацию, зададим аппаратную надежность для версий равной (0.99;0.95;0.90;0.80;0.70):

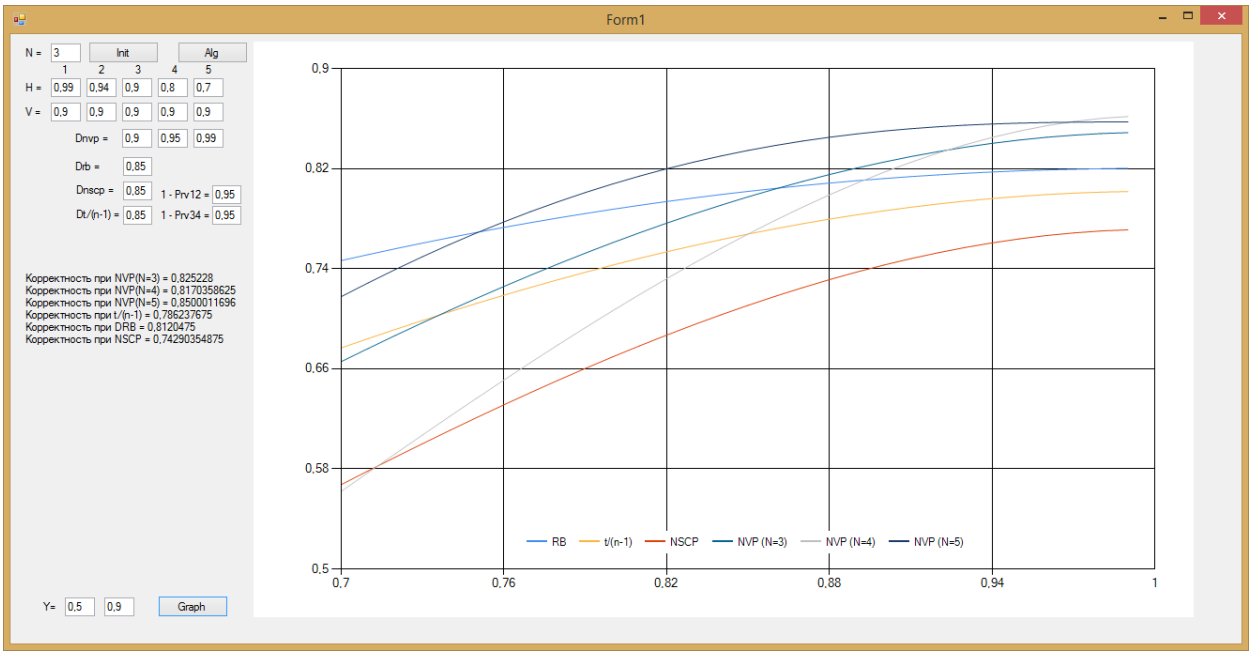


Рисунок 42 - Результат работы программы при различной аппаратной надежности

Сравним результаты, представленные на рисунке 42 с предыдущими, очевидно, что изменилось соотношение корректности версий, сместились точки пересечения, поскольку модель блоков восстановления использует только первые 2 надежные версии, а, к примеру, N-версионному программированию при N=5 приходится использовать и наименее надежные версии.

Выводы

В третьей главе предложен комбинированный селективный алгоритм для решения задачи выбора применяемых в блоке принятия решения моделей повышения отказоустойчивости ПО. Для оценки нижней границы надежности предложен модифицированный алгоритм расчета «деревьев сбоев», который ранее применялся для аппаратной надежности. Алгоритм расширен для возможности учитывать и аппаратные и программные сбои.

Полученная методика реализована программно и обеспечивает моделирование СУ с программной избыточностью, позволяя задавать все надежностные характеристики: вероятности сбоев каждой версии,

вероятности межверсионных ошибок, вероятность ошибки блока принятия решения, вероятность аппаратных сбоев, количество версий для NVP, вероятность возникновения связанной ошибки, влияющей на все версии, а также на решение, вызванное несовершенными спецификациями, а также минимальную надежность системы, которую необходимо достигнуть при обратном расчете.

Результаты работы данного инструмента позволяют оптимально скомпоновать разрабатываемую систему, так как на значение вероятности прохождения сбоя влияет не только характеристика каждого элемента системы, но и ее структура, поскольку от структуры зависят «срезы» - сочетания некорректной работы компонентов, достаточные для прохождения сбоя.

Для расчета верхней границы надежности предложена методика расчета параметра корректности.

Корректность - это сумма вероятности всех сочетаний возможных выходов мультиверсий, успешной работы аппаратного обеспечения и вариантов работы блока принятия решения, которые дадут корректный выход.

Для практической проверки предложенной методики и анализа результатов моделирования была реализована программная среда расчета величины корректности для различных мультиверсионных моделей, реализующая прямой и обратный расчет и построение графиков.

Предложенный селективный алгоритм, позволяет изучить влияние входных параметров надежности на ее корректность и вероятность прохождения сбоя, что является важным инструментом для разработчика отказоусточивых программных систем, поскольку позволяет осознанно выбрать методологию и определить требования к элементам разрабатываемой системы еще на этапе проектирования, что позволит упростить разработку системы с требуемым уровнем надежности и снизить затраты на ее реализацию.

Как показывают результаты моделирования, в большинстве случаев оптимальной оказывается модель мультиверсионного программирования, и возникает задача выбора алгоритма принятия правильного решения в ней.

4 Выбор алгоритма голосования для блока принятия решения мультиверсионной среды исполнения

На сегодняшний день все более актуальной становится задача создания отказоустойчивых систем управления, как для опасных и сложных производственных процессов, так и для автономных беспилотных объектов. И для всех этих задач требуется надежное программное обеспечение, как в целом для системы управления, так и для конкретных особо критичных к надежности процессов.

Наиболее эффективным способом повышения надежности ПО на сегодняшний день является подход с введением программной избыточностью - мультиверсионное программирование [108]. Но в программных системах нельзя просто дублировать версии, как это делается в аппаратном обеспечении, таким образом мы будем дублировать и все ошибки, как алгоритмические, так и ошибки кодирования, и не добьемся повышения надежности избыточной системы.

Состав мультиверсионой системы должен состоять из функционально эквивалентных, но алгоритмически различных версий, в идеальном случае, все версии должны разрабатываться различными разработчиками, на разных языках программирования, в разных средах разработки, с использованием разных библиотек, в случае такой необходимости [109]. Такой подход позволит минимизировать самый «опасный» тип ошибок – межверсионные или связные ошибки, это неправильные, но совпадающие между собой выходы версий. Данный тип ошибок наиболее опасен, поскольку его сложнее всего выявлять, поскольку в ситуации, когда, к примеру, 3 из 5 версий дали различные ошибочные выходы, возможно определить правильный выход для системы, отсеяв 3 ошибки, а в случае, когда ошибочные выходы равны, крайне сложно правильно принять решение.

Для решения того, какой же выход из множества версий признать верным и отправить на выход применяются различные алгоритмы [110], наиболее распространены алгоритмы голосования.

4.1 Базовые алгоритмы голосования

Голосование абсолютным большинством

В отказоустойчивой программной системе m-выходы из N, где N - число версий, а m - число согласных или количество совпадающих выходов, в соответствии с алгоритмом голосования, которое требуется для успеха системы [111]. Необходимо, чтобы за один вариант проголосовало абсолютное большинство версий, то есть $\lfloor N/2 \rfloor + 1$, где N – количество версий. Для примера с пятью версиями, нужно, чтобы за один вариант проголосовало не менее трех версий, иначе считается, что правильный ответ невозможно выбрать [112].

Два выхода из N-голосование

В [39] показано, что если пространство вывода велико и можно предположить истинную статистическую независимость вариантных отказов, то нет необходимости выбирать m больше 2, независимо от размера N. Термин «2 из N» голосование [113] используется для случая, когда число согласных m=2.

Очевидно, существует четкое различие между согласованностью и корректностью. Например, большинство алгоритмов предполагают, что если большинство выходов модуля «согласны», то основной вывод должен быть корректным [114]. Это, однако, может привести к ошибочным результатам, особенно, в чрезвычайном случае, когда количество возможных выходов модуля очень мало. Например, предположим, что выходные данные каждого модуля представляют собой одну переменную, которая принимает значения либо 0, либо 1. Это означает, что все ошибочные выходы автоматически согласуются [115], и очень вероятно, что если модули имеют ошибки, то может образоваться большинство согласованных ошибочных выходов.

В общем случае будет несколько выходных переменных, каждая из которых принимает несколько значений. Общее количество разрешенных комбинаций выходных переменных и их значений определяет число p выходных состояний программы или мощность выходного пространства. Если имеется несколько правильных выходов, то простой избиратель бесполезен. Следовательно, при голосовании мы будем считать, что для каждого входа есть только один правильный результат. Из этого следует, что при заданной выходной мощности p мы имеем одно правильное состояние выхода [116], а p - 1 состояния ошибки.

В этом контексте выбор среднего значения (или простое медианное голосование) представляет собой интересную и простую альтернативу вынесения решения, где медианой всех выходных значений выбран правильный ответ. Философия, лежащая в основе этого подхода, заключается в том, что в дополнение к быстродействию, алгоритм может обрабатывать несколько правильных ответов (а для малых выборок является менее предвзятым, чем усреднение или среднее значение голосования), и он, скорее всего, выберет значение, которое, по крайней мере, лежит в правильном диапазоне [117]. Этот метод был успешно применен в аэрокосмических приложениях.

Голосование согласованным большинством

Алгоритм голосования согласованным большинством – необходимо, чтобы за один вариант проголосовало большее число версий, необязательно чтобы количество голосов было больше половины от общего количества версий, для принятия решения достаточно, чтобы за определенный вариант проголосовало больше версий, чем за остальные [118]. В случае, если за несколько вариантов проголосовало одинаковое количество версий, выбирается любой из них, поскольку считается что они одинаково «корректны».

Если существует мажоритарное соглашение (m ≥ [(N + 1) / 2], N> 1), то этот ответ выбирается как корректный ответ.

В противном случае, если существует уникальное максимальное соглашение, но это число согласованных версий меньше [(N + 1) / 2], тогда этот ответ выбирается как корректный, в противном случае:

- Если согласованное голосование используется в N-версионном программировании [119], одна группа выбирается случайным образом, и ответ, связанный с этой группой, выбирается как корректный.

- Иначе, если согласованное голосование используется в модели согласованных восстанавливающихся блоков [120], все группы проходят приемочный тест, который затем используется для выбора правильного результата.

Согласованная стратегия голосования особенно эффективна в небольших выходных пространствах, поскольку она автоматически подстраивает голосование к изменениям эффективной мощности выходного пространства.

Можно показать, что при m ≥2 мажоритарное (абсолютным большинством) голосование дает верхнюю границу вероятности отказа системы с использованием согласованного голосования, а 2-выхода-из-N обеспечивает нижнюю границу [121].

Когда мощность выходного пространства равна 2, стратегия эквивалентна мажоритарному голосованию и голосованию 2-из-N, когда мощность выходного пространства стремится к бесконечности, при условии, что число согласных не меньше 2.

Нечеткое голосование согласованным большинством

По механизму голосования схож с прошлым алгоритмом, однако здесь добавлены элементы из теории нечетких множеств, каждая версия может голосовать за несколько близких ответов, но с разной степенью принадлежности – число от 0 до 1, определяющее «близость» к данному

значению, где 1 – равно значению, 0 – дальше от сравниваемого значения чем допуск E, а в промежутке – не равно значению, но находится не далее, чем на допуск E от него [122]. В итоге такого голосования версии получают различное и уже не целое число голосов, правильным или правильными (в случае совпадения числа голосов) признается версия с наибольшим числом голосов.

Основной проблемой применения соотношений равенства является строго заданное отклонение значений выходов, что не позволяет производить сравнения с точностью, превышающей ε. Применение нечетких множеств помогает разрешить эту проблему.

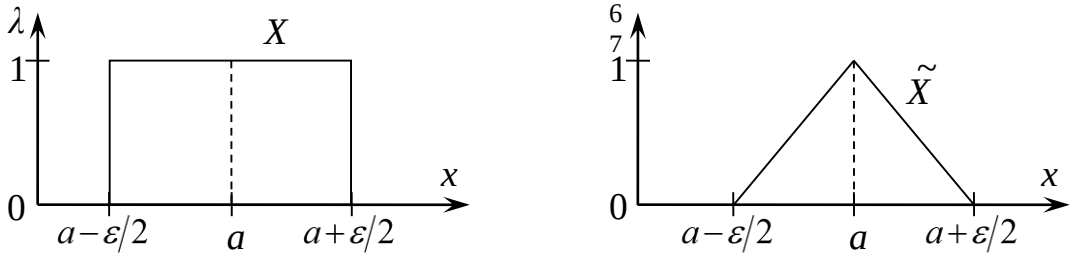
В четких множествах, элемент может либо быть элементом множества, либо не быть им. Пусть степень членства элемента во множестве задается некоторой функцией, которую будем называть характеристической функцией. Нечеткие же множества содержат элементы, которые обладают различной степенью принадлежности. Степень принадлежности задается числом на отрезке от 0 до 1, чем выше степень, тем ближе число к 1.

В классической теории алгоритмов голосования, используются соотношения равенства, основывающиеся на четких множествах [123]. Считается, что x равно некоторому числу a, если $|x - a| < \varepsilon$. Тогда характеристическая функция это равенство может быть представлена следующим образом:

$$\lambda_X(x) = \begin{cases} 1 & x \in (a - \varepsilon/2; a + \varepsilon/2) \\ 0 & x \notin (a - \varepsilon/2; a + \varepsilon/2) \end{cases} \quad (29)$$

Эта функция имеет прямоугольную форму (рисунок 43а) с центром в точке a и шириной основания равной ε. Степень принадлежности внутри этого прямоугольника равна 1, вне – равна 0. Однако, более удобно задавать степень принадлежности так, чтобы она уменьшалась по мере удаления от a. Например, задавать ее в виде треугольника (рисунок 43б):

$$\mu_{\tilde{X}}(x)=\begin{cases}1-\dfrac{|a-x|}{\varepsilon/2} & x\in(a-\varepsilon/2;a+\varepsilon/2)\\ 0 & x\notin(a-\varepsilon/2;a+\varepsilon/2)\end{cases} \qquad (30)$$



а) б)

Рисунок 43 - Характеристическая функция соотношения равенства x и a как а) четкого множества и б) нечеткого множества в форме треугольника

Матрица согласования в данном случае будет определяться как:

$$\text{R} = \{\text{rij}\}, \quad \text{где } r_{ij}=\mu_{\tilde{R}}(x_i,x_j)=\begin{cases}1-\dfrac{|x_i-x_j|}{\varepsilon} & |x_i-x_j|<\varepsilon\\ 0 & \text{иначе}\end{cases} \qquad (31)$$

Далее, производят усечение значений элементов полученной матрицы R к булевому виду $\bar{R}$, по некоторому заданному пороговому значению λ. Причем, значение λ нередко указывают в названии применяемой методики - например, для λ=0,5, название будет НГСБ-0,5.

Усечение значений происходит по следующей схеме:

$$\bar{r}_{ij}=\begin{cases}1 & r_{ij}\ge\lambda\\ 0 & r_{ij}<\lambda\end{cases} \qquad (32)$$

После этого, полученная матрица $\bar{R}=\{\bar{r}_{ij}\}$ используется как матрица согласования в четком методе голосования согласованным большинством (как базовом, так и взвешенном), рассмотренном выше.

Медианное голосование

В данном варианте считается, что выходы всех версий ошибочны и в качестве выхода берется усредненное значение всех выходов. Данный подход чаще применяется в случаях, когда невозможно напрямую сравнить выходы версий [124], например – когда выходами являются направление движения, вектора и т.д. Существуют взвешенные модификации медианного голосования, когда вклад каждой версии в ответ различен. Существуют различные реализации медианного голосования, в нашем случае происходит сортировка всех ответов и берется средний.

Данный метод голосования основан на том, что все выходы считаются ошибочными, а в качестве корректного результата принимается среднее между ними. Медианное голосование можно записать следующим образом:

$$r = \frac{1}{n}\sum_{i=1}^{n} \alpha_i \cdot x_i , \tag{33}$$

где r – корректный результат;

αi – весовой коэффициент;

xi – выход мультиверсии i;

n – количество мультиверсий.

Если αi ≠ 1, то данный алгоритм называется взвешенное медианное голосование.

Эти алгоритмы применяются в случаях, когда произвести прямое сравнение результатов мультиверсий затруднительно. Например, когда мультиверсионная система используется для поиска оптимальных направлений. В такой ситуации напрямую сравнивать векторы не имеет смысла, так как они как минимум разной длины.

4.2 Модификации существующих алгоритмов голосования

Как было отмечено выше, самыми опасными ошибками являются межверсионные [125], для повышения устойчивости к данным ошибкам

предложены модификации базовых алгоритмов голосования согласованным большинством и нечеткого голосования согласованным большинством.

Модификации заключаются во введении динамической оценки надежности каждой версии или ее «веса», имеющей так же элемент забывания. Вес считается как сумма результатов голосования, деленая на их количество. Технически это реализовано следующим образом - для каждой программной версии в системе создается булевый стек, заданной длинны, в который добавляется 0, в случае, если блок голосования решает, что версия дала неверный ответ и 1, если верный или, в случае нечеткого голосования, если принадлежность значения выхода версии к выигравшему классу >0, то есть все версии, прибавившие вес классу, выигравшему голосование будут помечены как верные, не зависимо от показателя принадлежности.

Поскольку длина стека фиксирована в рамках симуляции (задается на форме), новые данные будут замещать собой наиболее старые, то есть стек работает по принципу FIFO - первым пришел, первым ушел, это позволяет ввести элемент забывания на глубину стека - если, к примеру, глубина стека 100, то результаты работы версии старее, чем за 100 голосовании не будут учитываться. Элемент забывания нужен для обеспечения оперативной реакции системы на изменения поведения версий, в случае, когда версии могут существенно изменять свою надежность при изменении потока входных данных, необходимо оперативно изменить их рейтинг для наиболее корректного взвешенного голосовании [126].

Рейтинг определяется суммированием всех элементов стека, к примеру, если в стеке длинной 1000 элементов записано 993 значения «1» и 7 значений «0» вес данной версии будет равен 0,993.

При программной реализации внесено одно ограничение - вес версии не может быть равен единице, что может случаться на практике - достаточно надежные версии дают верный ответ 100, 1000, 10000 ит.д. раз подряд и без ограничения они получили бы весь стек из единиц (или TRUE), что дало бы им

рейтинг 1, таких ситуаций нельзя допускать, поскольку в случае неверного ответа данной версией, он получит вес 1, тогда как правильный ответ остальными N-1 версиями по весу лишь приблизится к 1 и проиграет голосование. К тому же аналитически, рейтинг надежности версии, равный 1, не имеет смысла, поскольку, если мы имеем абсолютно надежный программный модуль, смысл системы теряется. Таким образом, внесено ограничение при расчете рейтинга версии при каждом голосовании.

Также предложена нечеткая модификация t/(n-1) алгоритма, она заключается во введении нечеткой логики в компараторы. Поскольку компараторы имеют булевый выход, то они возвращают 0, если значения отличаются друг от друга на величину, не более заданного допуска, и 1, иначе. Разница значений, если она не превосходит допуск, не учитывается.

4.3 Программная реализация имитационной среды для сравнительного анализа и выбора алгоритмов голосования

В программе реализованы симуляции версий, которые работают соответственно заданным на форме параметрам – количество версий от 3 до 9, вероятность правильной работы при трех последовательных потоках данных для каждой из версий, длина наборов данных (соответственно, общее количество итераций равно трем длинам), вероятность появления межверсионной ошибки, вероятность возникновения неточности и допуск E. При каждом голосовании функция возвращает N ответов, с соответствующими для каждой версии и текущего набора входных данных вероятностями. Изменение надежности работы версий в процессе симуляции для трех наборов входных данных введена с целью исследования реакции системы на резкое изменение надежности работы версий, к примеру, если версия номер 3 имела надежность 0,97 в первом наборе, во втором надежность упала до 0,58, а в третьем вновь поднялась до 0,95.

В среде реализованы 5 алгоритмов принятия решения, это взвешенное голосование согласованным большинством с забыванием, его нечеткий вариант, медианное голосование и t/(n-1) – алгоритм принятия решения и его нечеткая модификация.

Рассмотрим процесс подробнее. При голосовании согласованным большинством блок голосования получает ответы симуляций версий, если значение выхода не совпадает с ранее полученным, создается новый класс, если значение совпадает с существующим классом, то происходит пересчет веса данного класса как

$$P_{итоговое} = P_{класса} + (1 - P_{класса}) \cdot P_{версии} \quad (34)$$

После того как все версии дали ответ происходит сравнения весов получившихся классов, выигрывает класс с наибольшим весом, как видно - это не всегда класс, за который проголосовало наибольшее число версий, значение имеет вес каждой версии, учитывается надежность каждой отдельной версии. После определения правильного выхода версии проголосовавшие за него получают в стек веса «1», а версии проголосовавшие иначе «0».

В данной модели отсутствует ограничение на время ответа версий, поскольку симуляции работают по одному алгоритму и нет смысла сравнивать их ресурсоемкость, однако при рассмотрении реальных версий, известных аппаратных ограничениях и требованиях ко времени реакции системы, есть смысл ввести подобное ограничение, чтобы учесть вероятность того, что ресурсоемкие версии могут не успеть дать ответ к моменту голосования, и не будут учитываться. Эти данные могут быть очень важными с случае построения систем, работающих в режиме реального времени.

В случае нечеткого голосования согласованным большинством после создания и оценки всех классов на основании выходов версий, программа производит еще один проход в процессе которого версии, значение выхода которых, не совпало со значением класса, но отличается от него не более

допуска E, а значит принадлежность классу равна > 0 также прибавляет вес классу и веса пересчитываются еще раз:

$$P_{итоговое} = P_{класса} + (1 - P_{класса}) \cdot P_{версии} \cdot K_{принадлежности} \quad (35)$$

$$K_{принадлежности} = 1 - (\left| X_{класса} - X_{версии} \right| / E), \quad (36)$$

, где X- значения класса и версии, давшей не совпадающий ответ, но укладывающийся в допуск E.

В рассматриваемой системе симуляции версий дают 3 типа ошибок: случайную ошибку, симулирующую сбой в модуле, межверсионную ошибку и неточность - ответ близкий к правильному, удаленный от него не более допуска, но не равный ему, этот тип ошибки симулирует «неточность» - ошибки округления при нехватке разрядности, неточности оцифровки выходов аналоговых датчиков и т.д., то есть ситуацию, когда алгоритмически версия сработала верно, но дала неточный ответ из-за ошибок округления, оцифровки, нехватки разрядности, большой разницы в порядке величин при операциях с плавающей запятой.

Ошибка проявляется с заданной для каждой версии и каждого потока данных вероятностью, в случае возникновения ошибки происходят следующие проверки – если это не первая ошибка в текущем голосовании, то с заданной вероятностью генерируется межверсионная ошибка – то есть возвращается значение, совпадающее со значением прошлой ошибки, эта ошибка симулирует связную ошибку – допущенный алгоритмический просчет, одинаковый в нескольких версиях, которые дадут одинаковую ошибку при одинаковом входе. Далее с заданной вероятностью генерируется «неточность» - возвращается выход не равный правильному, но удаленный от него не более чем на заданное отклонение E. Если предыдущие вероятности не срабатывают, то возвращается случайная ошибка, симулирующая сбой в текущей версии модуля.

Из результатов работы программы видно, что в отсутствие «неточностей» разницы в работе четкого и нечеткого вариантов методов голосования нет,

случайная ошибка всегда слишком «далеко» от правильного ответа, чтобы изменить вес классов. Из этого можно сделать вывод, если в системе нет возможных мест возникновения «неточностей» - оцифровка аналоговых сигналов, нехватка разрядности при математических операциях и т.д., то применение более ресурсоемкого алгоритма нечеткого голосования не даст преимуществ, однако, если возможность возникновения подобных «неточностей» имеется, то нечеткий алгоритм даст повышение надежности системы, единственным недостатком будет являться одинаковая оценка как версиям, давшим идеально правильный выход, так и версиям алгоритмически верным, но имеющим «неточности».

Реализация всех алгоритмов предполагает наличие блока принятия решений, осуществляющего процедуру голосования, который, в свою очередь, также реализован программно в имитационной модели.

В разработанной имитационной модели мультиверсии симулируются отдельной функцией [127], которая возвращает с заранее заданными вероятностями или правильный ответ или числовой результат одного из двух типов ошибок – «неточность» и «сбой», эти вероятности задаются в программе для 3 различных вариантов, что позволяет смоделировать различные реакции версий на изменение потока входных данных, когда после определенной итерации распределение вероятностей меняется и можно отследить реакцию системы на это, проверить корректность работы, скорость реакции в зависимости от выбранного алгоритма, длины стека и других параметров.

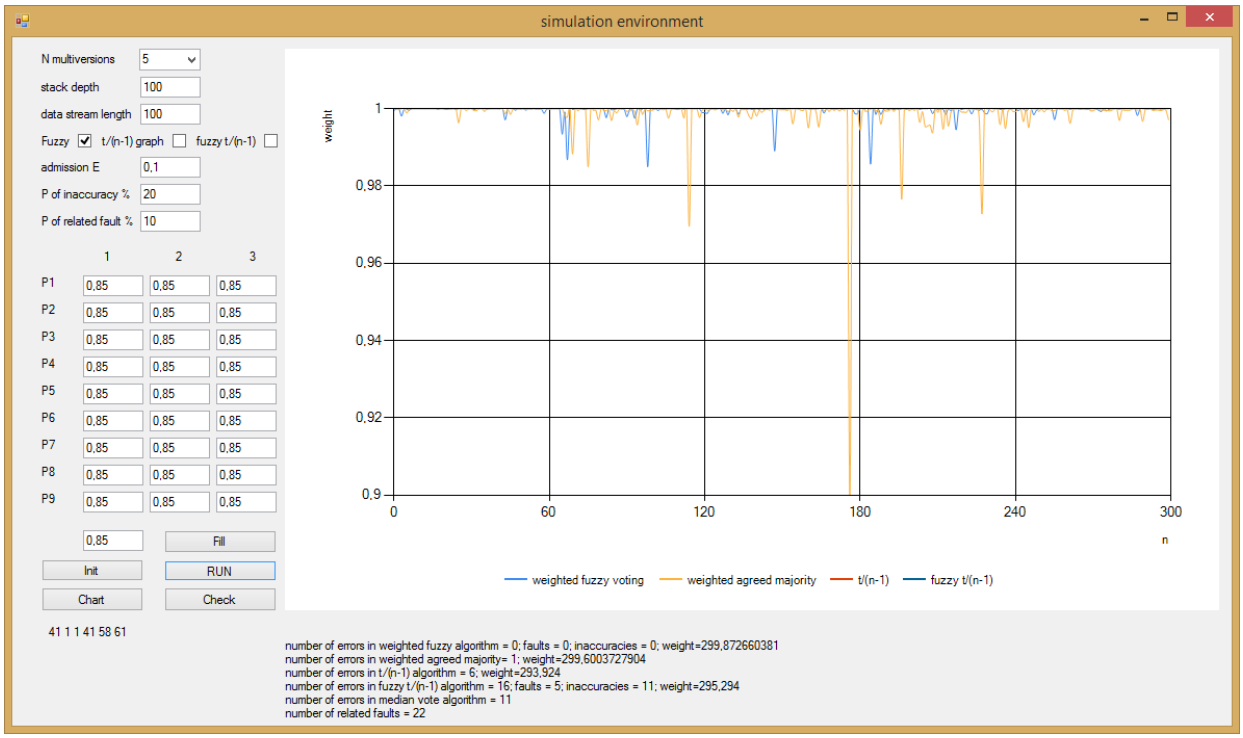


Рисунок 44 - Интерфейс имитационной среды моделирования

На рисунке 44 отображен интерфейс разработанной среды, а так-же видно, что при относительно надежных версиях даже в масштабе от 0.9 до 1 график весов решающего класса представляет собой практически прямую линию со значением 1, с несколькими незначительными «выбросами», что демонстрирует устойчивость системы к редким сбоям версий, данные сбой не оказывают влияния на работу системы, что так-же показывают результаты подсчета ошибок – ни одной для обоих алгоритмов голосования за все проходы.

Для удобства на форму выводится информация о последней зафиксированной ошибке на выходе блока голосования – номер итерации, значение выхода и вес данного класса. Система позволяет строить графики весов каждой версии и выигравших классов, по осям номера итерации и веса, имеется возможность изменять масштаб для наглядности, к примеру, при исследовании весов выигравших классов, их значения не опускаются ниже 0,9 и в масштабе от нуля до единицы представляют из себя практически ровную линию в области единицы, для лучшего восприятия в этом режиме масштаб изменяется на диапазон от 0,9 до 1 по оси веса.

По результатам моделирования выводится сумма ошибок за все итерации для каждого алгоритма, количество допущенных «неточностей» для нечетких алгоритмов и сумма возникших межверсионных ошибок. Если по результатам моделирования существует однозначно превосходящий другие алгоритм, система выводит сообщение о выбранном оптимальном алгоритме (рисунок 45).

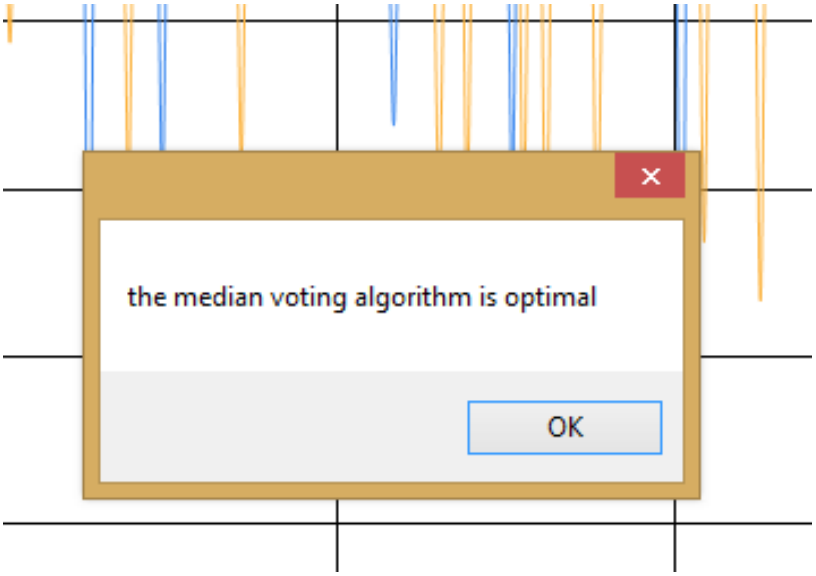


Рисунок 45 – Сообщение системы о выбранном оптимальном алгоритме

4.4 Результаты моделирования

Рассмотрим результаты исполнения программы с различными входными параметрами.

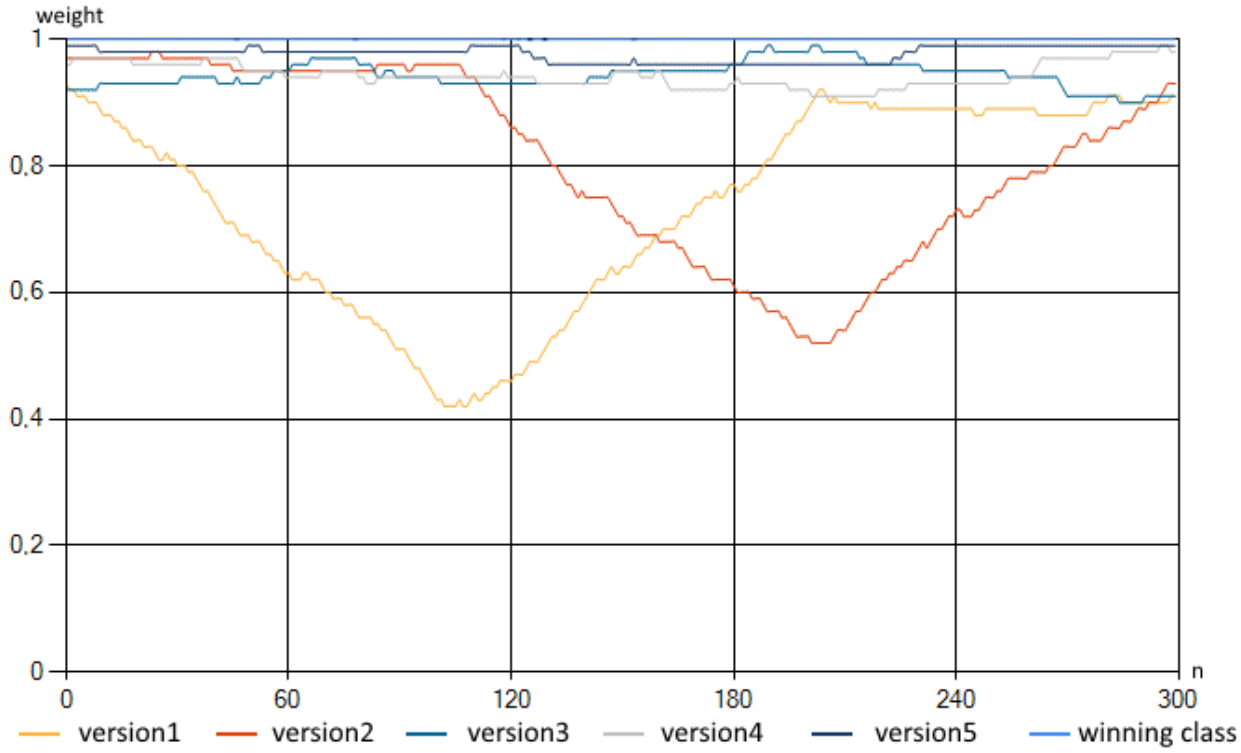


Рисунок 46 - Результаты работы имитационной модели реализации алгоритма голосования согласованным большинством (проявлен сбой в первой версии первого набора данных и во второй версии второго набора данных с 50% вероятностью).

На графике (Рисунок 46) видна реакция системы на изменение поведения версий, когда надежная версия начинает давать ошибки или наоборот, версия с низким весом перестает ошибаться, приведены графики для стека глубиной 100 и трех последовательных потоков данных по 100 голосований для каждого. Можно сделать выводы, что система достаточно быстро реагирует на изменения поведения версий и через количество голосований, равное глубине стека версии получают оценочные веса, достаточно точно соответствующие их вероятности правильного ответа.

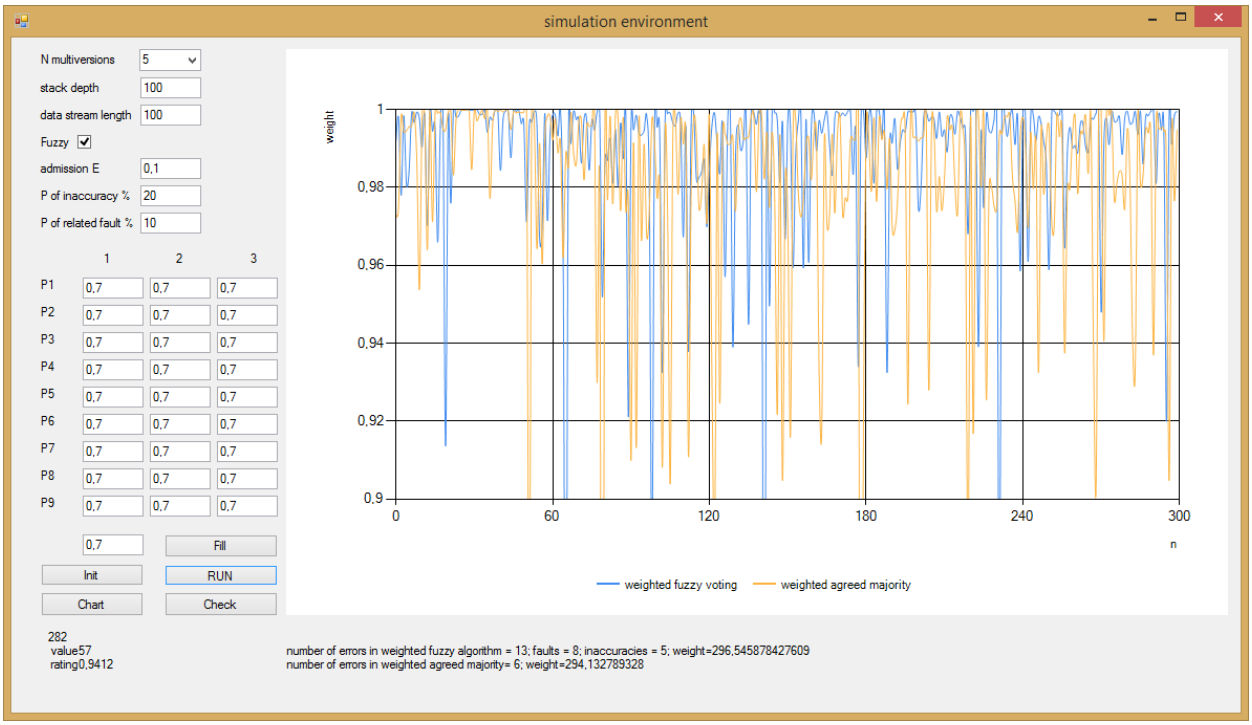


Рисунок 47 - Результаты моделирования при надежности всех версий равной 0.7 и N=5

На рисунке 47 отображена ситуация, смоделированная для наглядности, надежность всех версий задана равной 0.7, то есть каждая версия в 30% случаев вызова возвращает ошибку. На графике можно наглядно наблюдать реакцию системы на работу столь ненадежных программных компонент, и по выходу наблюдать наличие ошибок, все-таки прошедших на выход системы, однако, стоит отметить, что при 30% шансе на ошибку каждой отдельной версии, система с применением взвешенного алгоритма голосования согласованным большинством с забыванием допустила лишь 6 ошибок на 300 итераций, что более чем на порядок меньше частоты появления ошибок в версиях, данная ситуация показывает, насколько повышается надежность избыточной системы и эффективность предложенных алгоритмов. Исследуем поведение системы, при сохранении низкой надежности версий и увеличении их количества:

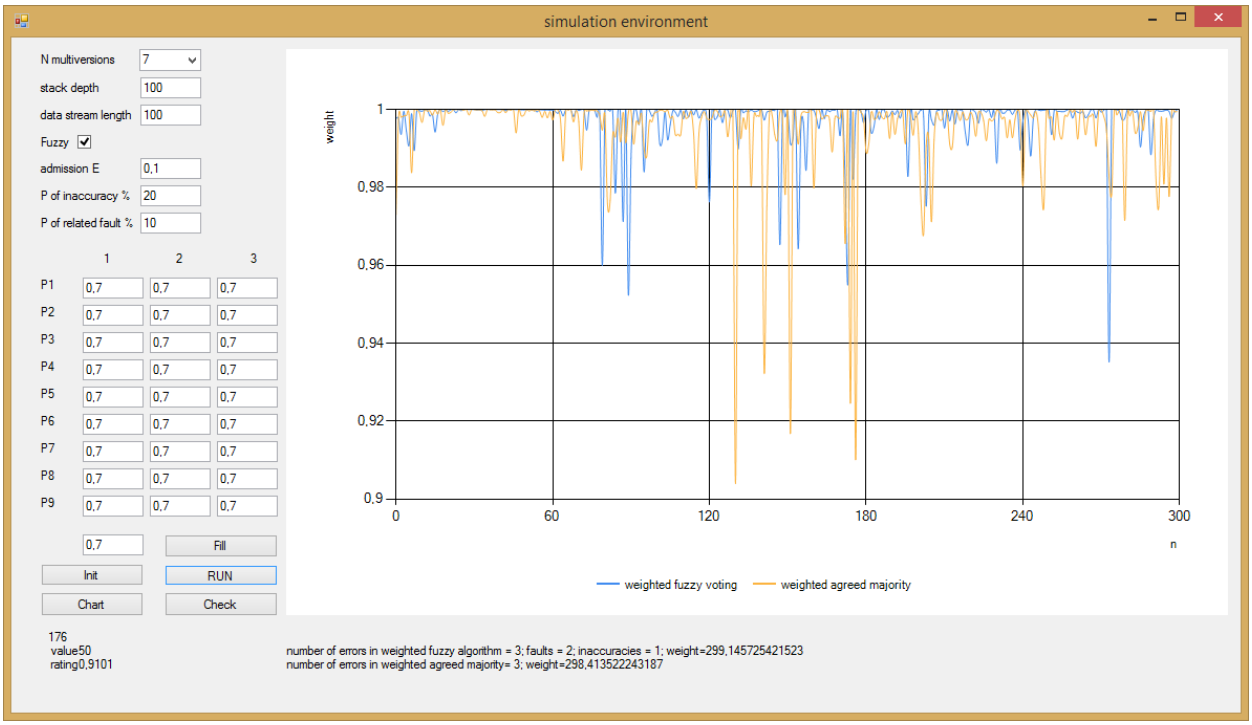


Рисунок 48 - Результаты моделирования при надежности всех версий равной 0.7 и N=7

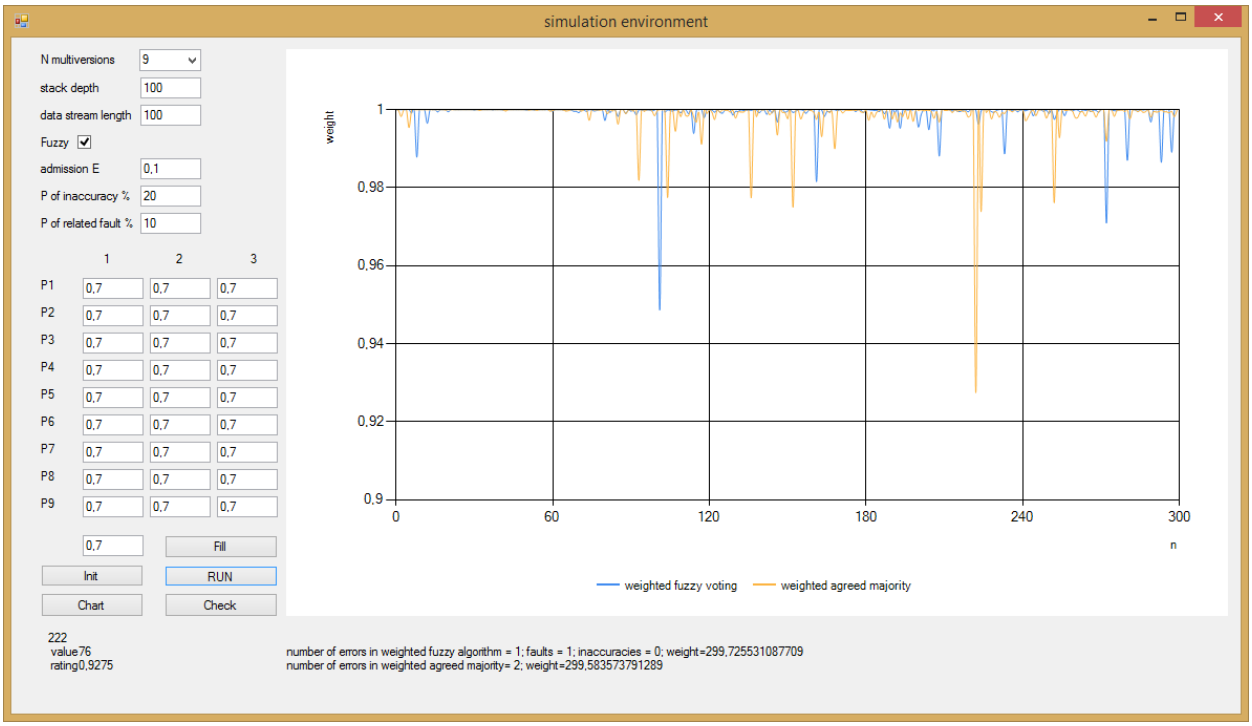


Рисунок 49 - Результаты моделирования при надежности всех версий равной 0.7 и N=9

Изучив результаты моделирования, представленные на рисунках 48 и 49, можно наблюдать, насколько возрастает надежность избыточной программной

системы, при сохранении столь-же не надежных версий (P=0.7), но при увеличении их количества. При 9 версиях предложенные алгоритмы дают лишь 1-2 ошибки за 300 итераций, при 30% вероятности ошибки каждой версии.

Сравним показатели надежности t/(n-1) алгоритма со взвешенными модификациями алгоритма голосования согласованным большинством, в его четком и нечетком варианте. Для этого проведем симуляцию в имитационной среде моделирования при следующих параметрах модели: количество итераций = 300, одинаковая надежность всех пяти версий во всех 300 прогонах, глубина стека = 100 (для взвешенных алгоритмов), допуск E = 0.1 (для нечеткой модификации), вероятность неточности P=20%, вероятность межверсионной ошибки P = 10%. Изменим надежность версий от относительно надежного значения 0.95 до 0.65 с шагом 0.05 и получим количество ошибок на 300 итераций для каждого алгоритма. Результаты моделирования представлены в таблице 2.

Таблица 2 – Результаты моделирования при различной надежности версий

Заданная надежность версий		0,65	0,7	0,75	0,8	0,85	0,9	0,95
Четкое голосование согласованным большинством	Кол-во ошибок	22	12	9	1	1	0	0
Нечеткое голосование согласованным большинством	Кол-во ошибок	24	14	7	3	1	0	0
	Из них сбоев	14	10	3	2	1	0	0
	неточностей	10	4	4	1	0	0	0
t/(n-1) алгоритм	Кол-во ошибок	39	24	17	10	3	1	0
Нечеткая модификация t/(n-1) алгоритма	Кол-во ошибок	44	35	34	21	16	4	3
	Из них сбоев	19	14	14	2	2	0	0
	неточностей	25	21	20	19	14	4	3
Медианное голосование	Кол-во ошибок	73	45	29	16	8	2	0
Межверсионных ошибок		134	94	66	41	26	10	1

Из результатов моделирования видно, что при относительно надежных версиях все алгоритмы обеспечивают работу без ошибок на протяжении 300 итераций, хотя каждая из 5 версий дает в среднем 15 ошибочных выходов за это время (при надежности 0.95 имитация версии выдаст в среднем 5 ошибок на 100 итераций). Исключение – нечеткий t/(n-1) алгоритм, результаты моделирования показывают, что нечеткая модификация очень неустойчива к возникновению неточностей, она не пропустила ни одного сбоя, однако при возврате неточностей первой или четвертой версией нечеткие компараторы возвращают совпадение с соседними версиями, давшими идеально правильный ответ и система отправляет на выход результат с неточностью.

При уменьшении надежности версий алгоритмы начинают допускать ошибки, однако в разном количестве, наиболее надежным оказался алгоритм согласованного голосования, точнее его взвешенная модификация с забыванием. Большее количество ошибок нечеткого алгоритма обусловлено случаями, когда ответом выбирается «неточность», ответ отдаленный от правильного не более чем на допуск E, но не равный ему, подобные ответы также засчитываются системой как ошибочные. t/(n-1) алгоритм начинает существенно уступать в надежности взвешенному алгоритму голосования согласованным большинством при относительно ненадежны версиях, поскольку все чаще возникают ситуации, в которых более t=2 версий дают неверный ответ, а в таких случаях корректная работа алгоритма не гарантируется.

Результаты так же показывают, что нечеткий вариант t/(n-1) алгоритма дает больше ошибок в целом, однако большая часть ошибок – это прошедшие неточности, количество сбоев даже меньше, чем у базового алгоритма.

Из представленного выше можно сделать вывод, что применение t/(n-1) алгоритма возможно при относительно надежных версиях, когда не будет возникать ситуаций одновременного сбоя более t версий. Его применение будет оправдано в ситуациях, когда необходимо снизить вычислительную

нагрузку на систему, особенно в случаях, когда компараторы выполнить просто, а создавать каждый раз множество классов и рассчитывать их веса слишком трудоемко (зависит от архитектуры системы).

Однако в случаях применения относительно ненадежных версий, или высокой вероятности межверсионной ошибки (когда несколько версий дадут одинаковый сбой одновременно) его применение нежелательно, поскольку в подобных ситуациях он показывает меньшую надежность по сравнению с алгоритмами мультиверсионного голосования. Относительно нечеткой модификации – ее применение оправдано в системах с вероятностью возникновения неточностей, прохождение которых не критично для системы, поскольку алгоритм часто допускает их прохождение, однако достаточно хорошо отсекает сбои.

Исследуем поведение системы с различными вероятностями возникновения неточностей, результаты представлены в таблице 3.

Таблица 3 – Результаты моделирования при различной вероятности возникновения неточности и P=0,8 для всех версий

Вероятность возникновения неточности		5%	10%	20%	50%	100%
Четкое голосование согласованным большинством	Кол-во ошибок	2	2	3	2	3
Нечеткое голосование согласованным большинством	Кол-во ошибок	2	3	7	8	5
	Из них сбоев	1	3	2	3	0
	неточностей	1	0	5	5	5
t/(n-1) алгоритм	Кол-во ошибок	8	10	13	11	11
Нечеткая модификация t/(n-1) алгоритма	Кол-во ошибок	10	12	25	30	69
	Из них сбоев	5	5	8	2	0
	неточностей	5	7	17	28	69
Медианное голосование	Кол-во ошибок	14	14	15	17	18
Межверсионных ошибок		48	41	44	43	50

Из результатов видно, что при повышении вероятности возникновения неточности нечеткие модификации алгоритмов принятия решения хоть и допускают прохождение данных неточностей, однако остаются устойчивыми к сбоям [56], и для выбора наиболее подходящего алгоритма необходимо понимать, насколько критично для системы прохождение именно неточностей.

К примеру – для многих систем поддержания курса с постоянной коррекцией неточности с незначительным отклонением от правильного значения не окажут негативного влияния на работу системы, в отличие от прохождения сбоев – когда система существенно изменит направление, что может привести к катастрофическим последствиям [128]. В случае систем, неустойчивым к неточностям, лучше применять классические варианты алгоритмов, поскольку они лучше отсеивают неточности, считая их неверными, вне зависимости от их близости к правильному ответу, в отличие от нечетких вариаций.

Таблица 4. Результаты моделирования при различной вероятности возникновения межверсионной ошибки и P=0,8 для всех версий.

Вероятность возникновения межверсионной ошибки		5%	10%	20%	50%	100%	100% при P=0,95
Четкое голосование согласованным большинством	Кол-во ошибок	2	3	4	5	13	0
Нечеткое голосование согласованным большинством	Кол-во ошибок	2	3	6	13	15	0
	Из них сбоев	1	2	2	10	11	0
	неточностей	1	1	4	3	4	0
t/(n-1) алгоритм	Кол-во ошибок	4	7	10	16	21	1
Нечеткая модификация t/(n-1) алгоритма	Кол-во ошибок	10	17	19	23	22	7
	Из них сбоев	3	4	6	12	13	2
	неточностей	7	13	13	11	9	5
Медианное голосование	Кол-во ошибок	14	15	19	16	17	0
Межверсионных ошибок		26	41	66	204	392	31

Результаты моделирования показывают значительно большую устойчивость алгоритмов голосования к межверсионным ошибкам, по сравнению с t/(n-1) алгоритмом. Так же добавлен дополнительный столбец в таблицу, чтобы показать, что при достаточно надежных в целом версиях (P=0.95 для всех), даже 100% межверсионных ошибок не оказывают влияния на надежность системы с предложенными модифицированными алгоритмами голосования, чего нельзя сказать о t/(n-1) алгоритме, который допускает сбои даже при надежных версиях.

Интересно отметить, что межверсионные ошибки не оказывают влияния на работу медианного алгоритма голосования, поскольку в нем не происходит сравнения ответов версий для изменения весов классов, а коллекция ответов просто сортируется во величине значений. Для медианного голосования нет разницы, совпадающие ошибки выдают версии или нет. Рассмотрим пример – если 5 версий дали ответы (3;3;6;19;19) и ответы «3» и «19» являются межверсионными ошибками, он все равно выберет ответ «6», как средний. Это же свойство его работы делает его устойчивым к выбросам, в отличии от реализации алгоритмов голосования с усреднением выходов, если одна или несколько версий даст ответ, отличающийся на несколько порядков в любую сторону, это не повлияет на работу алгоритма.

Результаты, полученные с помощью разработанной имитационной среды моделирования, показывают эффективность подхода с введением программной избыточности, также доказывают возможность создания надежной системы из ненадежных программных модулей. Что же касается t/(n-1) алгоритма, который представляет интерес, как альтернатива алгоритмам голосования, его применение оправдано лишь в системах с существенными ограничениями по вычислительным ресурсам и применением достаточно надежных версий, поскольку алгоритм показывает худшую устойчивость к ненадежным версиям и различным типам ошибок.

Выводы

В четвертой главе исследованы базовые алгоритмы голосования, сделан вывод о недостаточной устойчивости существующих алгоритмов к межверсионным ошибкам. Предложены модификации базовых алгоритмов голосования согласованным большинством и нечеткого голосования согласованным большинством, необходимые для повышения надежности блока принятия решения. Модификации заключаются во введении динамической оценки надежности каждой версии или ее «веса», имеющей элемент забывания.

Модифицированные алгоритмы были реализованы в программном инструменте – имитационной среде, с целью их проверки и сравнения эффективности.

Особый интерес представляет нетривиальная модель t/(n-1)-версионного программирования, предложенная Цзе Сюй (Jie Xu) основанная на t/(n-1) диагностируемости.

В работе впервые предложена нечеткая модификация t/(n-1) алгоритма, она заключается во введении нечеткой логики в компараторы. Поскольку компараторы имеют булевый выход, то они возвращают 0, если значения отличаются друг от друга на величину, не более заданного допуска, и 1, иначе. Разница значений, если она не превосходит допуск, не учитывается.

Рассмотрены результаты исполнения имитационной среды моделирования с различными входными параметрами. Произведен сравнительный анализ показателей надежности t/(n-1) алгоритма по сравнению с взвешенными модификациями алгоритма голосования согласованным большинством (его четкой и не четкой версии) и медианным голосованием.

5 Практическая реализация инструментальных средств повышения отказоустойчивости программных комплексов систем управления реального времени

5.1 Имитационная среда выбора методологии повышения отказоустойчивости системы с программной избыточностью

На основании сравнительного анализа в предыдущих главах сделан вывод о том, что наиболее эффективными являются именно мультиверсионные модели, однако среди них нет однозначно лучшей, выбор конкретной зависит от требований к системе и характеристик составляющих ее модулей, как программных, так и аппаратных [129]. Возникает задача обоснованного выбора модели повышения отказоустойчивости.

Для решения задачи выбора, на этапе принятия решения, необходимо сравнение различных моделей в одинаковых условиях, с обязательной возможностью исследования в рамках работающей среды исполнения отказоустойчивого ПО.

Не всегда именно методология N-версионного программирования является наиболее подходящей [130], например, существуют задачи, для которых крайне сложно реализуются алгоритмы выбора из коллекции ответов, но возможна реализация приемочного теста. Для большинства задач возможна реализация всех основных методологий повышения отказоустойчивости путем введения программной избыточности [131]. Возникает задача выбора наиболее оптимальной методологии, либо по соотношению характеристик и затрат на разработку, либо по целевой характеристике [132], если не все методологии позволяют достичь требуемого значения. В любом случае необходим инструментарий, позволяющий оценить характеристики надежности системы или ее компонента для реализации всех основных мультиверсионных методологий при заданных характеристиках программных модулей и блока принятия решения.

Методика позволяет сравнить все алгоритмы на одинаковом наборе ответов версий на каждой итерации, что позволяет сравнить результаты работы всех методологий в одинаковых условиях, соответствующих заданным характеристикам системы. Это позволяет понять какая из методологий будет лучше работать именно в представленных условиях: при определенной надежности версий на каждом из изменяющихся потоков данных, с какой вероятностью будут возникать межверсионные ошибки или ошибки округления, с какой вероятностью будут проявляться сбои в приемочных тестах и т.д.

Для проверки предложенной методики и анализа результатов моделирования может быть использована программная реализация [133] среды исполнения различных мультиверсионных моделей.

Предложен программный комплекс, который позволит исполнять модули не только с помощью методологии N-версионного программирования (NVP), а также с использованием остальных распространенных мультиверсионных моделей [134]: восстанавливающихся блоков (RB), согласованных восстанавливающихся блоков (CRB), t/(n-1) версионного программирования и мультиверсионного программирования с самопроверкой (NSCP).

В данной среде программными модулями будут выступать симуляции версий, они работают в соответствии с заданными на форме характеристиками: Общая надежность версии в каждом потоке данных (в системе реализовано 3 последовательно изменяющих потока данных, для исследования поведения системы на изменение работы программных модулей при различных входных данных), вероятности возникновения межверсионной ошибки, «неточности», параметр допуска E для нечетких алгоритмов голосования используется симуляциями как значение разброса «неточностей».

В рассматриваемой системе симуляции версий дают 3 типа ошибок: случайную ошибку, симулирующую сбой в модуле, межверсионную ошибку

и неточность - ответ близкий к правильному, удаленный от него не более допуска, но не равный ему, этот тип ошибки симулирует «неточность» - ошибки округления при нехватке разрядности, неточности оцифровки выходов аналоговых датчиков и т.д., то есть ситуацию, когда алгоритмически версия сработала верно, но дала неточный ответ из-за ошибок округления, оцифровки, нехватки разрядности, большой разницы в порядке величин при операциях с плавающей запятой [135]. Ошибка проявляется с заданной для каждой версии и каждого потока данных вероятностью, в случае возникновения ошибки происходят следующие проверки – если это не первая ошибка в текущем голосовании, то с заданной вероятностью генерируется межверсионная ошибка – то есть возвращается значение, совпадающее со значением прошлой ошибки, эта ошибка симулирует связную ошибку – допущенный алгоритмический просчет, одинаковый в нескольких версиях, которые дадут одинаковую ошибку при одинаковом входе.

Далее с заданной вероятностью генерируется «неточность» - возвращается выход не равный правильному, но удаленный от него не более чем на заданное отклонение E. Если предыдущие вероятности не срабатывают, то возвращается случайная ошибка, симулирующая сбой в текущей версии модуля.

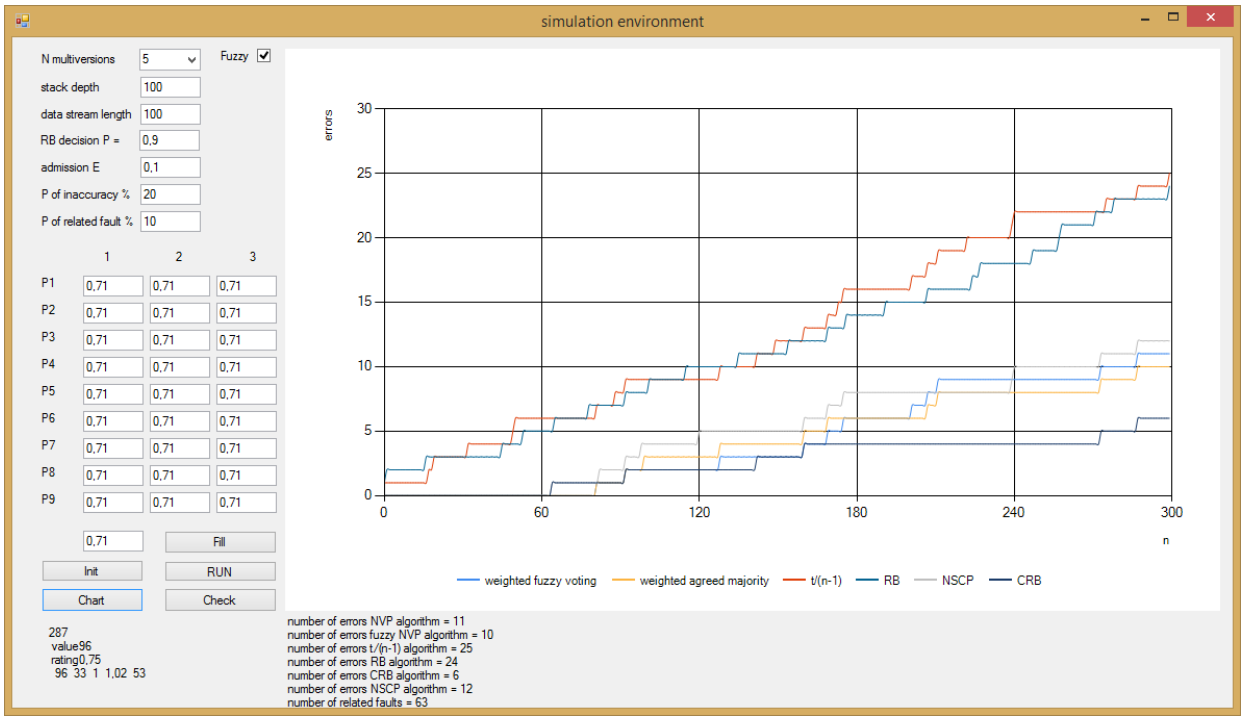


Рисунок 50 – Результат моделирование, количество ошибок в каждой методологии и график их распределения по времени.

На рисунке 50 представлен результат работы разработанной среды, в варианте с симуляциями версий. На форме задается: количество версий, глубина стека для взвешенных алгоритмов с забыванием, длина потока данных, вероятность ошибки приемочного теста алгоритмов с восстанавливаюшимися блоками, допуск вхождения для нечетких алгоритмов, вероятности возникновения неточностей и межверсионных ошибок, надежности каждой версии в каждом из трех потоков данных, кнопки: заполнения значений надежности версий, для удобства, инициализации значений параметров системы, запуска, отрисовки графика и проверки, окно графика, а так же окно вывода информации о последней возникшей ошибке и результаты моделирования – общее количество накопленных ошибок за все итерации и количество межверсионных ошибок. Графики отображают распределение возникновения ошибок алгоритмов по времени.

Для практической проверки работоспособности предложенного инструментария разработана программная среда, в которой в качестве версий используются 5 различных алгоритмов оптимизации [136]: метод деления

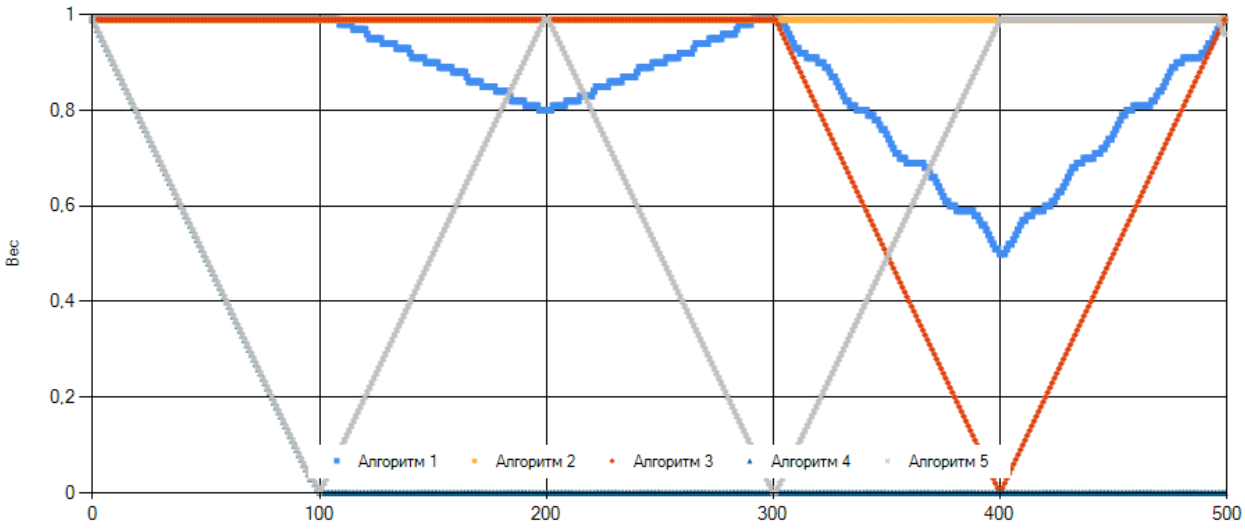
отрезка пополам, метод золотого сечения [137], метод дихотомии [138], метод квадратичной аппроксимации и метод Фибоначчи [139]. В каждом методе наложено ограничение на количество итераций K=10, поскольку в системах реального времени нам нужно гарантировано получить ответ за заданный промежуток времени [140], нельзя допускать случаем слишком длительного исполнения циклов, тем более будет интереснее сравнить точность алгоритмов, которую они смогут достигнуть при введенном ограничении, одинаковом для всех версий.

Система может принимать решение на основе любой из рассмотренных ранее методологий, однако на примере будем рассматривать выходы N-версионного программирования с нечетким взвешенным голосованием согласованным большинством с забыванием, выбор нечеткого голосования обусловлен характером выходов версий – ожидаются ответы близкие к правильному, но не обязательно равные им, потому лучше использовать алгоритмы с элементом нечеткой логики, чтобы близкие ответы могли увеличивать веса классов друг друга, повышая устойчивость системы.

Выбор взвешенного алгоритма позволяет нам численно сравнить проверочные алгоритмы, по результатам прогонов мы будем накапливать их «веса» - посчитанную оценку вероятности правильного ответа, основанную на статистике работы каждой версии, чем чаще версия будет ошибаться, тем меньше будет ее «вес», оптимальной же будет версия с большей суммой весов. Соответственно, оптимальным будет алгоритм, который реализует данная версия, с наибольшей суммой весов.

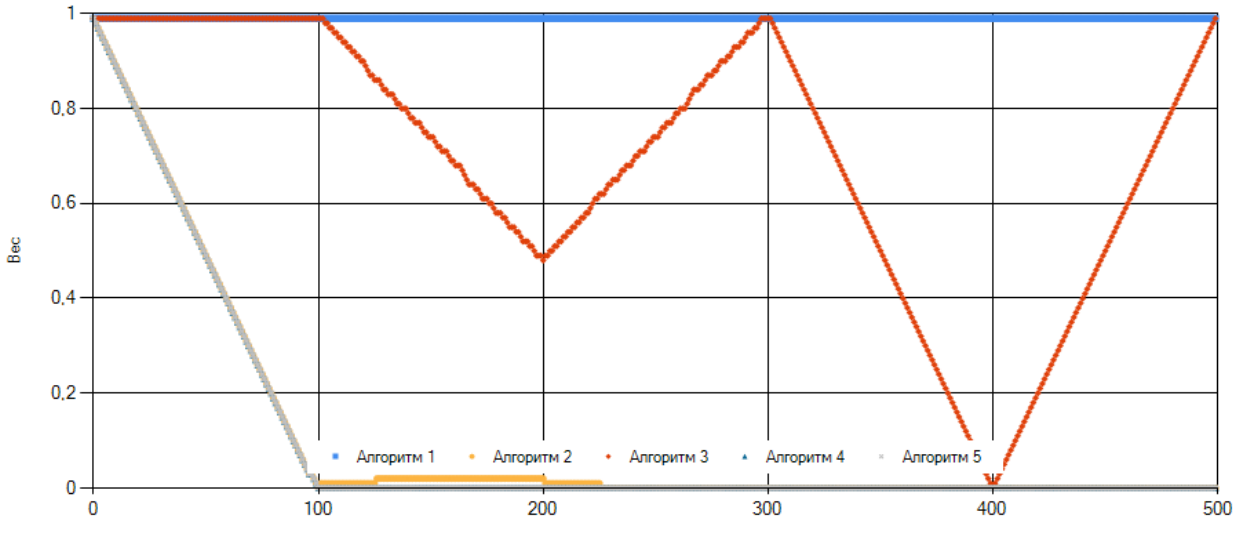
В соответствии с введенными параметрами система осуществляет мультиверсионное исполнение всех алгоритмов и принятие решения, каждую итерацию меняются коэффициенты функции, так же каждый 100 итераций меняется тип функции, в программе реализовано 5 различных функций. Это сделано для исследования работы проверочных алгоритмов, поскольку на различных типах функций разные проверочные алгоритмы показывают

различные результаты. Итоговая оценка алгоритмов представлять собой сумму результатов работы по всем 5 функциям со ста различными коэффициентами для каждой, что является достаточно объективным показателем и позволяет отсеять алгоритмы, работающие хорошо только в узком диапазоне параметров [141].



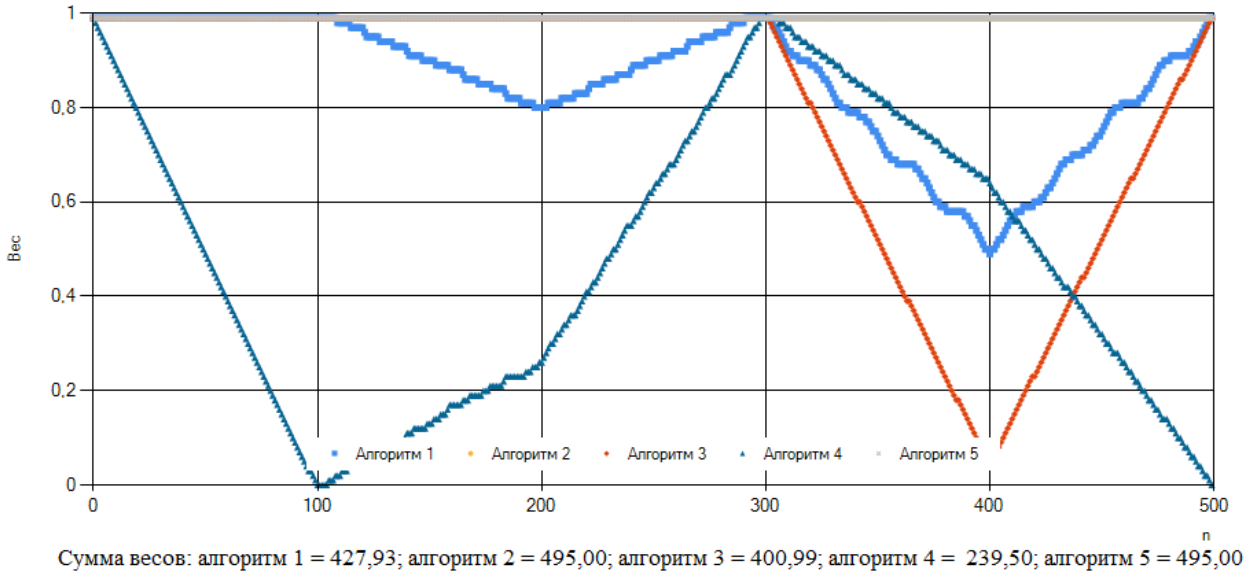
Сумма весов: алгоритм 1 = 428,91; алгоритм 2 = 495,00; алгоритм 3 = 396,99; алгоритм 4 = 49,53; алгоритм 5 = 296,97

Рисунок 51 - Результат работы системы при E = 0.045



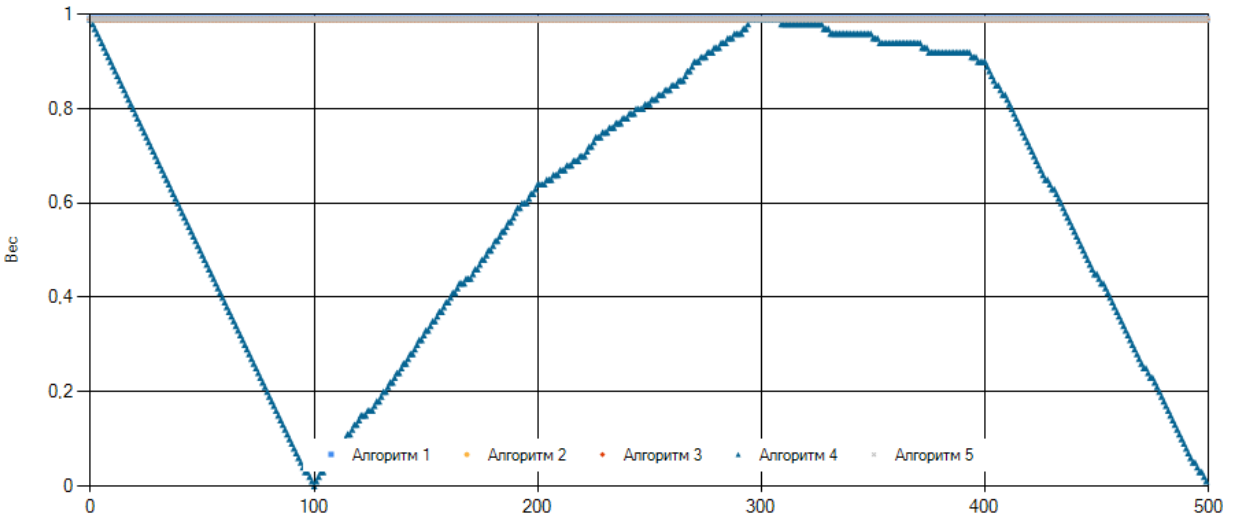
Сумма весов: алгоритм 1 = 495,00; алгоритм 2 = 51,53; алгоритм 3 = 346,98; алгоритм 4 = 49,53; алгоритм 5 = 49,53

Рисунок 52 - Результат работы системы при E = 0.001



Сумма весов: алгоритм 1 = 427,93; алгоритм 2 = 495,00; алгоритм 3 = 400,99; алгоритм 4 = 239,50; алгоритм 5 = 495,00

Рисунок 53 - Результат работы системы при E = 3



Сумма весов: алгоритм 1 = 495,00; алгоритм 2 = 495,00; алгоритм 3 = 495,00; алгоритм 4 = 303,45; алгоритм 5 = 495,00

Рисунок 54 - Результат работы системы при E = 10

Рассмотрим результаты работы системы на рисунке 51, при допуске 0.045 есть однозначно лучший алгоритм - второй, он всегда голосует за выигравший класс, то есть он не обязательно равен ему, но отличается не более чем на 0.045 в любую сторону, не обязательно иметь максимально близкий к правильному ответ, в соответствии с логикой работы взвешенного алгоритма голосования 1 в стек весов получают все версии с коэффициентом вхождения больше нуля, вне зависимости от его конкретного значения.

Очевидно, что величина допуска E будет достаточно существенно влиять на работу системы, поскольку определяет отклонение, которое может допустить версия и быть признанной верно ответившей, рассмотрим случаи уменьшения E до 0.001 на рисунке 52 и его увеличения до значений 3 и 10 на рисунках 53 и 54 соответственно. Из результатов видно, что при уменьшении допуска суммы весов резко падают, поскольку алгоритм начинает все «строже» принимать решение о корректности выхода версии, помещая в стек весов 0 за отклонение, превышающее 0.001.

При увеличении допуска наблюдается обратная картина, суммы весов растут, достигая максимального показателя в 495 уже для нескольких версий, поскольку даже алгоритмы дающие ответы с существенным отклонением признаются ответившими верно. В таком случает система уже не может однозначно выбрать наилучший алгоритм и выдает список оптимальных, то есть получивших максимальные суммы весов.

Разработанная среда позволяет сравнить все основные методологии повышения отказоустойчивости с введением программной избыточности в одинаковых условиях, с заданными характеристиками системы и вероятностями корректной работы всех компонент, получить по результатам моделирования характеристики системы с применением данных методологий. Это упрощает выбор методологии для реализации разрабатываемого ПО.

Программная реализация с реальными алгоритмами позволяет выбрать оптимальный алгоритм на заданном наборе функций и величине допустимого отклонения, а также показывает работоспособность предложенного инструментария на реальной прикладной задаче, где система уже не знает заведомо правильного выхода, а работает в реальных условиях, а не с симуляциями версий.

5.2 Практическая реализация мультиверсионной среды исполнения реального времени

Для практической проверки и подтверждения работоспособности выбранных с помощью имитационной среды моделей и алгоритмов разработана мультиверсионная среда исполнения реального времени на базе ОСРВ FreeRTOS v10. В среде реализован блок принятия решения, выделенные очереди, функции, реализующие предложенные модели и алгоритмы, которые запускаются в виде отдельных потоков и другие модификации ОСРВ, необходимые для обеспечения требуемой функциональности [142].

Наиболее рациональный способ создания мультиверсионной среды исполнения реального времени – взять за основу существующую операционную систему реального времени (ОСРВ), с открытым исходным кодом и возможностью его модификации и использования для собственных нужд, в том числе и коммерческих. Самостоятельная разработка ОСРВ с нуля, ее портирование на множество архитектур, отладка и тестирование, дальнейшее развитие системы потребуют очень больших материальных и временных затрат [143], не предоставляя значимых преимуществ перед использованием существующих систем.

Рассмотрим технические особенности реализации подобных программных модулей – необходимые модификации обеспечения требуемой функциональности, а именно – возможности мультиверсионного исполнения отказоустойчивого ПО в реальном времени [144].

В первую очередь нам необходимо создать блок принятия решения, который также будет исполняться в виде потока в ОСРВ, запускать версии и т.д.

Блок принятия решения реализован в виде функции *static void RunComp()* в начале которой объявляются локальные переменные, выделяется память, при необходимости вызываются функции инициализации массивов и т.д. Если данные необходимо прочитать из файла и их размер не слишком велик для

размещения в оперативной памяти, то файл открывается, данные сохраняются в переменных и файл закрывается, это более разумно, чем читать по одному значению каждый раз [145]. После этого запускаются потоки версий командами *xTaskCreate(version1, "version1", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 5, &ver1);* .

Сами версии представляют собой подобные функции *static void version1(),* в начале которых тоже объявляются локальные переменные и выделяется память, основным отличием является основной функциональный код, он заключен в бесконечный цикл *for (;;)* первой строкой в котором расположена команда *vTaskSuspend(NULL);* , таким образом, при запуске потока версии происходит объявление переменных, выделение памяти и все подготовительные процедуры, после чего версия уходит в состояние Suspend, из которого будет вызываться при необходимости, подготовленная к вычислению задачи.

Для передачи входных данных в версии и получения результатов работы лучше всего использовать механизм очередей, поскольку он реализует ряд функций, защищающих корректность данных и повышающих надежность [146].

Таким образом, в начале функционального кода внутри бесконечного цикла будет чтение входных данных из очереди, а в конце – запись полученных результатов в очередь. Функцию return нельзя использовать, поскольку в качестве потоков используются функции типа *static void,* и при корректном исполнении функция никогда не доходит до конца .

После того, как потоки версий запущены и находятся в режиме Suspend, при необходимости исполнения очередной итерации данной задачи входные данные помещаются в соответствующие очереди и очереди вызываются командой *vTaskResume(ver1);*, после чего версиям передается исполнение, они продолжают выполнять код, следующий за *vTaskSuspend(NULL);*, получают необходимые данные из очереди, производят вычисления, помещают

результаты в очередь, переключаются на следующую итерацию цикла и снова первой командой в нем срабатывает *vTaskSuspend(NULL);*, отправляя версию в Suspend.

Далее в коде блока принятия решения можно проверить, сколько версий уже предоставили ответы командой *uxQueueMessagesWaiting(QueueName);*, которая вернет длину очереди результатов вычислений. После чего можно считать ответы всех версий командами *xQueueReceive*, которые также очистят очередь при считывании. Используя полученные результаты производится мультиверсионное голосование, и ответ, признанный верным отправляется на выход.

В данном случае используется взвешенный алгоритм голосования согласованным большинством, он позволит не только определить правильный ответ из набора ответов версий, но и позволит сравнить работу самих версий по сумме их весов. В случае жесткого ограничения на время ответа, если одна из версий не успевает предоставить ответ, ее можно завершить принудительно, осуществив голосование с использование полученных в срок ответов версий [147].

Программная реализация среды исполнения РВ

Реализована мультиверсионная среда исполнения реального времени для проверки работоспособности предложенных в статье решений. Среда реализована в ОСРВ FreeRTOS v10, в ней реализован блок принятия решения, выделенные очереди, функции, реализующие предложенные модели и алгоритмы, которые запускаются в виде отдельных потоков.

В первой реализации перенесены алгоритмы оптимизации из имитационной среды, для сравнения имитационной среды и реальной реализации в ОСРВ FreeRTOS [148]. С результатами исполнения можно ознакомиться на рисунке 55.

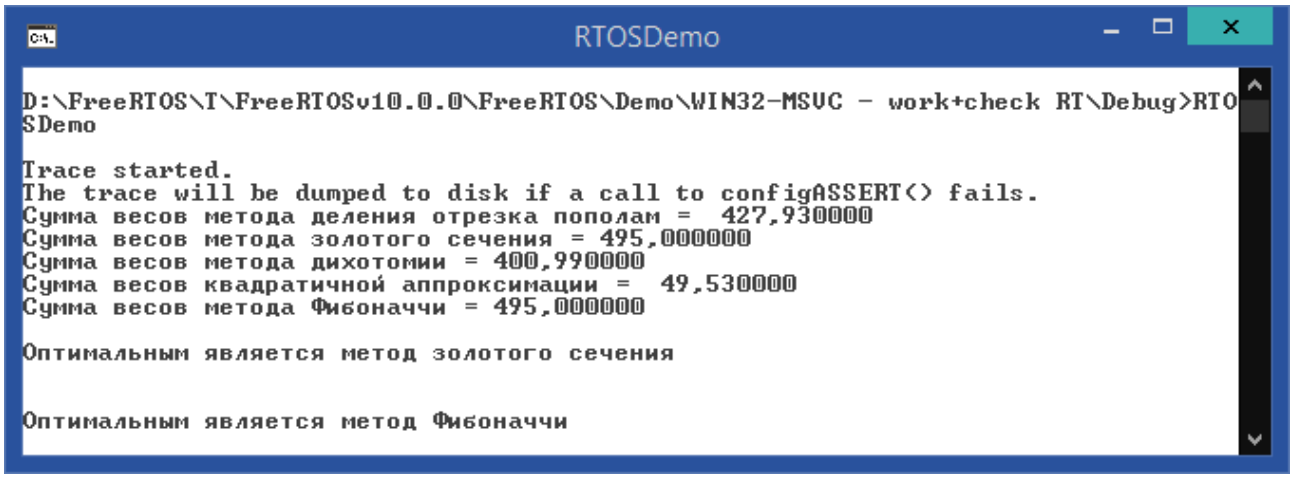


Рисунок 55 – FreeRTOS с мультиверсионным исполнением алгоритмов оптимизации.

Полученные результаты совпадают с результатами имитационной среды, что подтверждает корректность предложенных моделей и позволяет верифицировать программные реализации [149].

Далее в среде исполнения в качестве версий были реализованы алгоритмы управления, а именно – 3 различных алгоритма распознавания голосовых команд. Система использовалась для решения прикладной задачи – повышения эффективности системы распознавания голосовых команд для передачи управляющих воздействий автономному беспилотному объекту. В качестве подобных объектов может выступать как моторизованное инвалидное кресло, мультироторные системы, так и вспомогательные устройства для работ в открытом космосе. Использовались выходы трех готовых алгоритмов распознавания, на основе которых система выбирала правильный ответ.

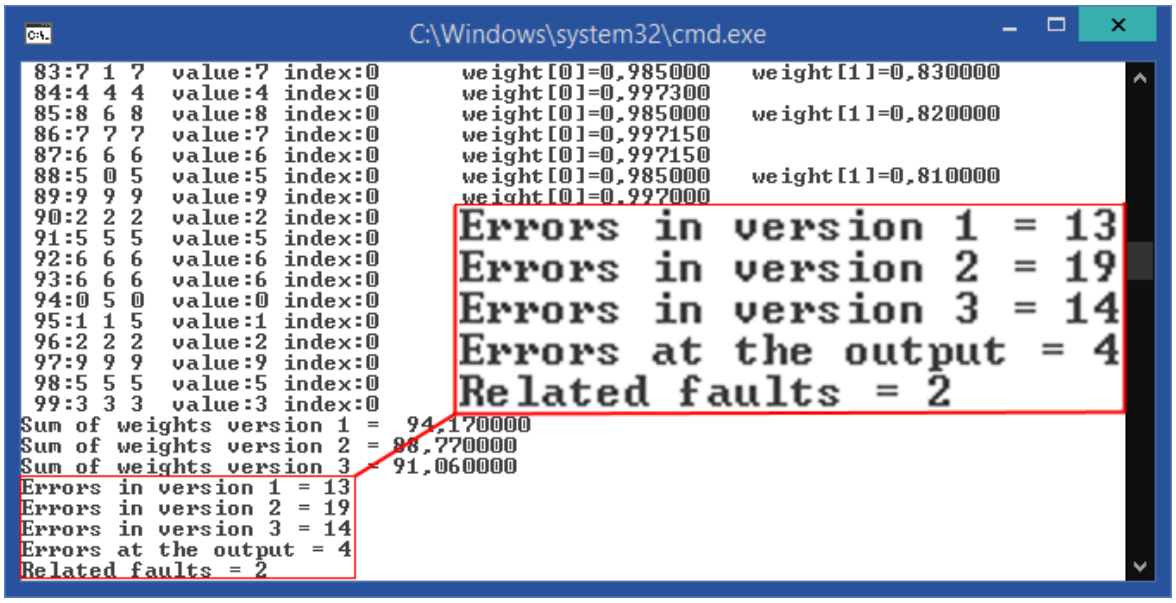


Рисунок 56 – FreeRTOS с мультиверсионным исполнением алгоритмов распознавания голосовых команд.

Результаты работы программной системы (рисунок 56) показывают эффективность мультиверсионного подхода, поскольку качество работы системы превышает качество любого из используемых алгоритмов (процент корректно распознанных команд для первого алгоритма составил 87, для второго алгоритма - 81, для третьего алгоритма - 86, а система корректно распознала 96% команд). Исполнение происходило в режиме реального времени [150], результаты подтверждают выполнение требований по времени реакции системы на событие. Каждая итерация мультиверсионного голосования уложилась в 1 мс.

Описанные модификации позволяют создать на базе ОСРВ FreeRTOS мультиверсионную среду исполнения реального времени. Что предоставляет новый инструментарий для решения достаточно актуальной на сегодняшний день проблемы создания отказоустойчивых систем управления реального времени [151]. Практическая реализация подтверждает корректность предложенного подхода, работоспособность всех технических решений и применимость системы для исполнения алгоритмов управления. Проверка на

реальной прикладной задаче подтверждает эффективность мультиверсионного подхода и корректность работы системы в целом.

5.3 Деэвтрофикационная модель предсказания времени наработки на отказ

Предыдущие рассуждения иллюстрируют тесное взаимодействие между устранением сбоев и прогнозированием сбоев и поддерживают их сбор в однократную проверку. Это несмотря на то, что проверка часто ограничивается удалением сбоя и связана с одним из основных действий, применимых к устранению сбоев, а именно верификации [152].

Помимо явного выделения необходимости для валидации процедур и механизмов отказоустойчивости, учитывая устранение сбоев и прогнозирование сбоев, как двух составляющих одной и той же деятельности – валидации, представляют большой интерес, поскольку это приводит к лучшему пониманию термина охвата и, таким образом, важной проблемы, которая возникает в результате описанной выше рекурсии: проверки валидации или того, как достичь уверенности в методах и инструментах, используемых для укрепления доверия к системе.

Здесь охват относится к измерению показателя репрезентативности ситуаций, в которых система представлена во время ее валидации, по сравнению с реальными ситуациями, с которыми она столкнется в течение своего эксплуатационного периода. Несовершенство охвата усиливает связь между устранением сбоев и прогнозированием сбоев, поскольку можно считать, что необходимость прогнозирования сбоев проистекает из несовершенного охвата устранения неисправностей.

Проанализировав рассматриваемые ранее модели надежности программного обеспечения, можно сделать вывод о том, что наиболее подходящими являются модели, основанные на экспоненциальном законе распределения времени до нахождении следующей ошибки при тестировании. Иными словами – с каждой выявленной и исправленной ошибкой в отлаживаемом программном обеспечении время нахождения следующей

ошибки растет по экспоненциальному закону. Эта модель наиболее применима в общем случае, поскольку не требует знания о внутренней структуре программы, которая не всегда известна при использовании закрытых библиотек и модулей, а требует для расчетов лишь данные о времени нахождения ошибок при тестировании программной системы [153].

Для решения задачи оценки времени наработки на отказ программного обеспечения предлагается деэвтрофикационная модель. Разумеется, зависимость выражена не в простейшем виде - $T = e^n$, где T – время ожидания (n+1)-ой ошибки, n-количество выявленных ошибок. Очевидно, что необходимо ввести поправочные коэффициенты, которые будут различаться для каждой системы, в зависимости от ее характеристик (размеры кода, язык программирования, используемые алгоритмы, среда исполнения, качество разработки и т.д.). Расчет данных коэффициентов, выявление закономерности в распределении экспериментальных данных и расчета разброса от искомой функции распределения, представляет из себя научно-практическую задачу.

Разработана среда, загружающая из внешнего файла данные – экспериментальные результаты тестирования программного обеспечения, в данном случае – время нахождения ошибки. Для примера рассмотрим экспериментальные данные о результатах тестирования программного обеспечения - CSR1.DAT - 397 time-between-failures data. Chapter 4 [6].

Из данных, представленных на рисунке 57 можно сделать вывод о том, что реальное распределение времени ожидания ошибок - точки на графике, не расположены на идеальной экспоненте, т.е. имеют разброс, чего и следовало ожидать. Однако, путем несложных манипуляций, можно убедиться в правильном характере распределения. Для решения поставленной задачи предложена следующая методика. Первым шагом необходимо подобрать коэффициент λ для экспоненциальной функции $T = e^{\lambda * n}$, в нашем случае подходит значение 0,02. Полученная экспонента проходит через всю область распределения экспериментальных точек.

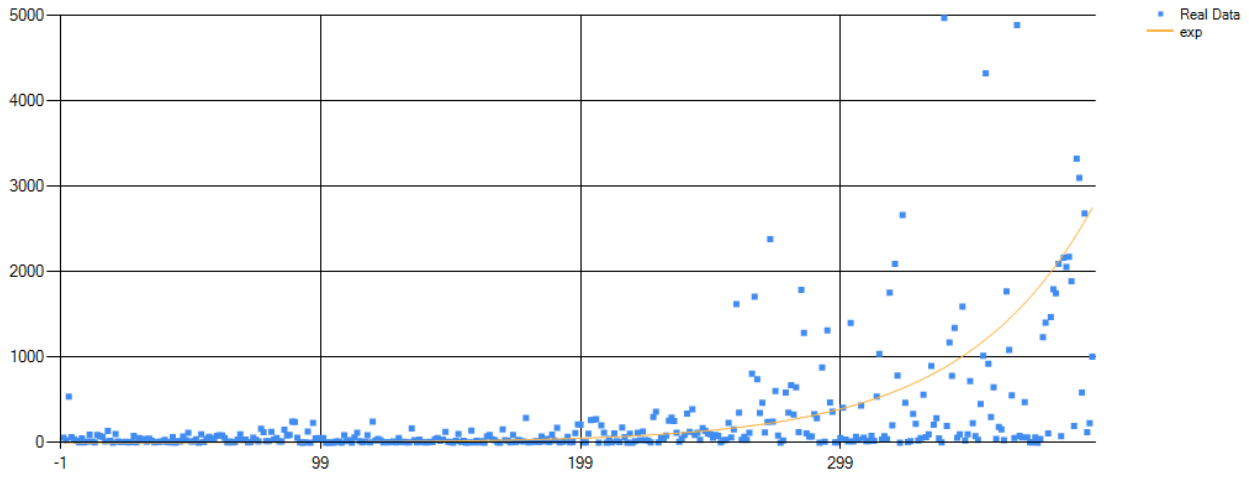


Рисунок 57 - Экспериментальные данные до обработки

Далее задаем допустимый разброс и коридор, тем самым определяем точки, не являющиеся «выбросами», в данной модели используется двойная проверка – отклонение в абсолютных числах в 200 с и отклонение от текущего значения экспоненты в 60% (параметры можно изменить по желанию на форме). В итоге мы получаем 2 массива точек – исходный и результат нашего отбора, не содержащий выбросов, в данном случаем отбор прошло 74% точек.

Вторым шагом применим сглаживание к обоим массивам, получим сглаженные кривые, которые можем наблюдать на рисунке 58.

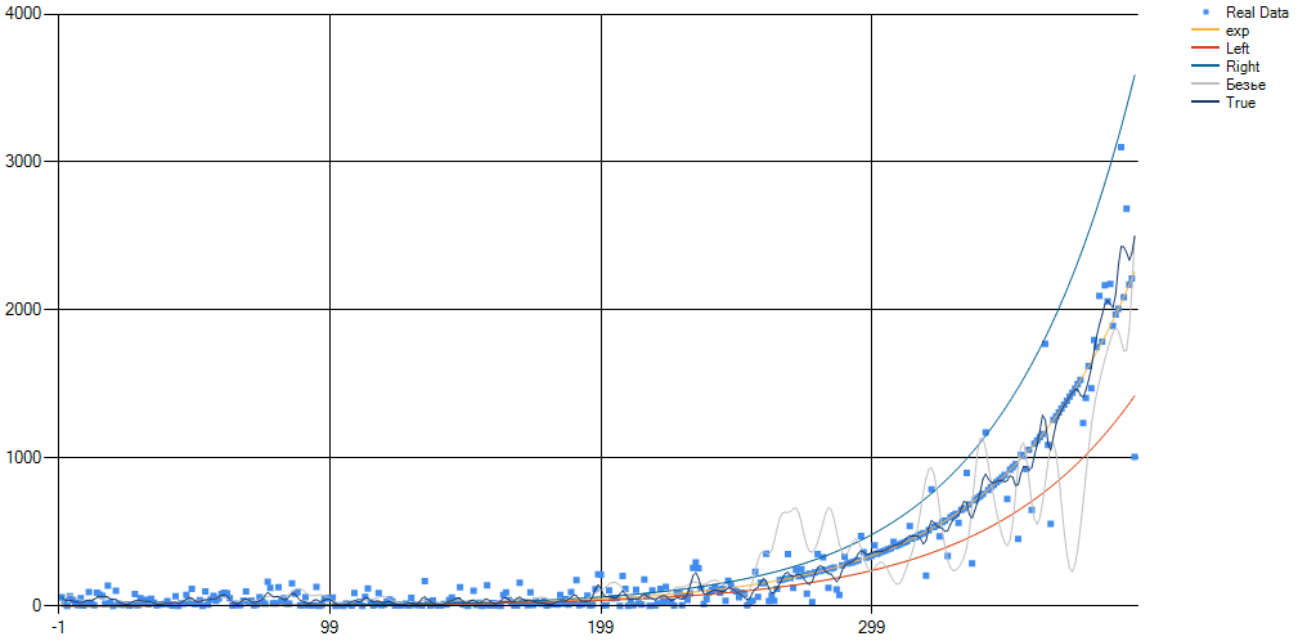


Рисунок 58 - Результат работы программы

Серым цветом выделена кривая, построенная по точкам, содержащим выбросы, по ней сложно судить о характере распределения. Темно-синяя линия - результат сглаживания по точкам, не содержащим выбросы, по ней уже можно сделать вывод, что распределение действительно очень близко к экспоненциальному и закон распределения соблюдается.

По этой сглаженной кривой в 200 точках рассчитываем и усредняем коэффициент λ, чтобы получить экспоненту, максимально приближенную к реальным точкам. Результат расчета $\lambda = 0{,}0195$, применим его к построению экспоненты и при тех-же параметрах модели, таким образом, процент не отброшенных точек вырос до 75. Теперь, использую экспоненту с рассчитанным коэффициентом, мы можем предсказать дальнейшее поведение системы.

Мы имеем значение экспоненты, являющееся математическим ожиданием времени безотказной работы, рассчитанный «коридор» вхождения, который представляет собой дисперсию и процент вхождения точек – нашу вероятность корректности предсказания. Необходимо отметить, что мы можем менять параметры системы – увеличивая «коридор» мы уменьшаем точность предсказания, поскольку увеличиваем дисперсию, при этом повышаем вероятность попадания в эту область, а уменьшая коридор, мы напротив, увеличиваем точность, но теряем в значении вероятности правильного предсказания – реальное значение с меньшим шансом попадет в более узкую область.

Важно отметить, что тестирование в реальном времени (когда количество входных данных и, следовательно, выходных в единицу времени совпадает с реальным режимом эксплуатации системы) актуально для систем с относительно малым сроком службы ПО – беспилотные летательные объекты с небольшой дальностью автономного полета, системы корректировки траекторий движения модулей (ступеней) ракетной техники и т.д.

Для систем с длительным сроком эксплуатации необходимо ускорить процесс тестирования, чтобы получить гарантированное время наработки на отказ, измеряемое в годах, не расходуя на этап тестирования и отладки столь значительные отрезки времени. Этого можно достичь тестированием не в масштабе реального времени, к примеру, если система при эксплуатации будет получать 20 команд в секунду, этого быстродействия достаточно, а условия эксплуатации вносят жесткие ограничения на потребляемую мощность, тепловыделение, массо-габаритные характеристики аппаратного обеспечения и т.д. Тогда как при тестировании можно обеспечить производительность в 200, 2000, 20000 и более команд в секунду, получив соответствующий коэффициент ускорения относительно реальной эксплуатации системы. Это позволит существенно сократить временные затраты на тестирование и получить прогнозируемое время наработки на отказ (MTTF), превосходящее само время тестирования.

Выводы

Для практической проверки разработана имитационная среда, реализующая все рассмотренные мультиверсионные модели и модифицированные алгоритмы голосования в них. Результаты моделирования показывают адекватность подхода и позволяют сравнить основные мультиверсионные модели по количеству допущенных ошибок в одинаковых условиях, а также позволяют сравнить различные алгоритмы принятия решения в мультиверсионных системах.

Реализована мультиверсионная среда исполнения реального времени для проверки работоспособности предложенной во второй главе типовой структуры. Среда реализована в ОСРВ FreeRTOS v10, в ней реализован блок принятия решения, выделенные очереди, функции, реализующие предложенные модели и алгоритмы, которые запускаются в виде отдельных потоков.

Система использовалась для решения прикладной задачи – повышения эффективности распознавания голосовых команд для передачи управляющих воздействий на подсистемы автономных беспилотных объектов. Объектами применения разработанного программно-аппаратного комплекса на базе FreeRTOS служат такие автономные беспилотные объекты, как моторизованные и мультироторные системы, а также вспомогательные устройства для работ в открытом космосе. Использовались выходы 3 готовых алгоритмов распознавания, на основе которых система выбирала правильный ответ.

Результаты работы программной системы показывают эффективность мультиверсионного подхода, поскольку качество работы системы превышает качество любого из используемых алгоритмов (процент корректно распознанных команд для первого алгоритма составил 87, для второго алгоритма - 81, для третьего алгоритма - 86, а система корректно распознала 96% команд). Исполнение происходило в режиме реального времени, результаты подтверждают выполнение требований по времени реакции системы на событие. Каждая итерация мультиверсионного голосования уложилась в 1 мс.

В рамках реализации прикладных задач, на этапе тестирования возникает ряд проблем, связанных с невозможностью гарантировано выявить 100% ошибок в сложном программной комплексе.

На основе модели Джелинского – Моранды и полученном экспоненциальном распределении времени обнаружения ошибок в тестируемом ПО предложена деэвтрофикационная модель предсказания времени наработки на отказ (MTTF). Модель реализована в виде программного инструмента, проведено исследование экспериментальных данных о результатах тестирования программного обеспечения (397 точек):

Данная модель и реализованный инструмент позволяют предсказать среднее время наработки на отказ, его разброс и вероятность попадания в

предсказанный диапазон. Также данный инструмент позволяет существенно ускорить тестирование, поскольку дает возможность тестировать систему не в масштабе реального времени, а настолько быстрее, насколько позволит производительность тестовой платформы, что позволит гарантировать время наработки на отказ в месяцах, затрачивая на тестирования на порядки меньшие промежутки времени.

Заключение

1. Сформирована типовая структура системы управления, в том числе ее программной составляющей, которая позволяет: разработать имитационную среду моделирования, в которой можно протестировать отдельные программные модули, получить их характеристики и характеристики системы при их применении; реализовать мультиверсионную среду исполнения реального времени на основе ОСРВ, применимую на большинстве современных одноплатных компьютеров и платах разработчиков.

2. Предложен комбинированный селективный алгоритм для выбора оптимальной модели повышения отказоустойчивости, основанной на программной избыточности, использующей следующие модели:

2.1 Модифицирована и реализована модель деревьев сбоев для возможности расчета характеристик системы с учетом и аппаратных и программных модулей. Полученный инструмент является полезным для разработчика на практике, поскольку позволяет получить обоснованную компоновку системы, а также изучить влияние структуры системы и характеристики ее модулей на общие характеристики надежности системы.

2.2 Расчетный параметр корректности впервые предложен и представляет собой сумму всех срезов, по аналогии с деревьями сбоев, однако учитывающий, в отличии от них, все возможные сочетания работы узлов, которые приведут к корректному выходу. Реализован программный инструмент, автоматизирующий расчет параметра, как прямой, позволяющий получить характеристики системы, так и обратный, позволяющий определить минимальные требования к надежности модулей.

3. Предложены новые взвешенные модификации с элементом забывания алгоритмов голосования, которые повышают устойчивость к межверсионным ошибкам и повышающие надежность в целом. Реализована программная среда, реализующая предложенные алгоритмы, которая позволила доказать их

эффективность по сравнению с базовыми алгоритмами (на 0,08 сократилась вероятность прохождения ошибки).

4. Разработана имитационная среда моделирования, реализующая все распространенные модели повышения отказоустойчивости и нетривиальный алгоритм t/(n-1)-вариантного программирования, которая позволяет сравнить все модели в одинаковых условиях – на одинаковых выходах симуляций версий, работающих в соответствии с заданными параметрами надежности.

5. Реализована мультиверсионная среда исполнения реального времени на основе ОСРВ FreeRTOS v10, для проверки предложенной технологии. Результаты работы практически реализованной системы, решающей реальную задачу, показывают применимость предложенной технологии и эффективность моделей и алгоритмов.

6. Впервые предложена и реализована деэвтрофикационная модель предсказания времени наработки на отказ. Модель позволяет предсказать параметры надежности программного модуля по результатам его испытаний и существенно (на порядки, в зависимости от производительности тестовой платформы) ускоряет этап тестирования.

Таким образом, цель данного исследования достигнута путем создания моделей, алгоритмов и программных инструментов, применяемых на различных этапах жизненного цикла создания отказоустойчивых программных комплексов встраиваемых систем управления реального времени.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Каштанов, В.А. Теория надежности сложных систем // А.И. Медведев, В.А. Каштанов. – М.: Издательская фирма «Физико-математическая литература», 2010 г. 608 с.

2. Сарамуд М.В., Ковалев И.В., Лосев В.В., Посконин М.В., Калинин А.О., К вопросу классификации автономных беспилотных объектов // Актуальные проблемы авиации и космонавтики – 2017. Том 2, С. 64-66.

3. Ковалев И.В., Лосев В.В., Сарамуд М.В., Ковалев Д.И., Петросян М.О. К вопросу реализации мультиверсионной среды исполнения бортового программного обеспечения автономных беспилотных объектов средствами операционной системы реального времени // Вестник Сибирского государственного аэрокосмического университета им. академика М.Ф. Решетнева. 2017. Т. 18. № 1. С. 58-61.

4. Kulyagin V.A., Tsarev R.Yu., Prokopenko A.V., Nikiforov A.Yu., Kovalev I.V. N-version design of fault-tolerant control software for communications satellite system // 2015 International Siberian Conference on Control and Communications (SIBCON), 2015, С. 1 – 5

5. Селищев А.А. Повышение надёжности программного обеспечения // В сборнике: Наука сегодня фундаментальные и прикладные исследования материалы международной научно-практической конференции. 2016. С. 47.

6. R. Lyu, Michael. Handbook of Software Reliability Engeneering / Michael R. Lyu. IEEE Computer Society Press.USA. 1995. 851 с.

7. Непомнящий В.А. Надежность атомных электростанций //Надежность и безопасность энергетики. 2013. № 21. С. 2-11.

8. Надежность функционирования и информационная безопасность телекоммуникационных систем железнодорожного транспорта // материалы Всероссийской научно-технической Интернет-конференции с международным участием / Омский государственный университет путей сообщения, редактор В. Е. Митрохин. 2013.

9. Лахин О.И., Полников А.С., Симонова Е.В., Скобелев П.О. Теория сложности и проблема управления жизненным циклом изделий аэрокосмической промышленности // Информационно-управляющие системы. 2015. № 1 (74). С. 4-12.

10. Исхаков Ш.Ш., Ковалев Ф.Е., Косенков Р.Э., Мохнаткин А.П. Проблемы оценивания надёжности и безопасности эксплуатируемых сооружений наземной космической инфраструктуры и идентификации их технических состояний // Известия Петербургского университета путей сообщения. 2016. Т. 13. № 4 (49). С. 592-599.

11. Севастьянов Н.Н. Управление надёжностью космических аппаратов с длительными сроками эксплуатации // Космонавтика и ракетостроение. 2017. № 3 (96). С. 133-148.

12. Артюхова А.Л. Надежность программного обеспечения информационных систем // В сборнике: Информационные системы и технологии материалы докладов II международной научно-технической заочной конференции «ИСТ-2016». Юго-Западный государственный университет. 2016. С. 39-40.

13. Энергетика: Эффективность, Надежность, Безопасность // Материалы XX Всероссийской научно-технической конференции / Томский политехнический университет. 2014. Том 1

14. Соболев С.А. Методы обеспечения сбое- и отказоустойчивости аппаратуры космического назначения // Вопросы атомной науки и техники. Серия: Физика радиационного воздействия на радиоэлектронную аппаратуру. 2012. № 2. С. 17-21.

15. Ляшенко А.В., Игнатьев А.А., Проскуряков Г.М., Поздняков М.В. Отказоустойчивые пилотажно-навигационные комплексы пилотируемых и беспилотных летательных аппаратов // Гетеромагнитная микроэлектроника. 2016. № 21. С. 59-67.

16. Тюгашев А.А. Подход к обеспечению отказоустойчивости космических аппаратов на основе автоматизации проектирования интеллектуальных бортовых программных средств // Надежность и качество сложных систем. 2016. № 4 (16). С. 106-112.

17. Карпов А.П., Попова М.С. Влияние программного обеспечения на надежность информационной системы // Аллея науки. 2016. № 4. С. 710-713.

18. Sterner, B.J. Computerized Interlocking System — A Multidimensional Structure in the Pursuit of Safety // IMechE Railway Engineer International, 1978. С. 29-30.

19. Frullini, R., Lazzari, A. Use of Microprocessor in Fail-Safe on Board Equipment // Proceedings of the International Conference on Railway Safety Control and Automation Towards 21st Century. London, U.K., 1984. С. 292-299.

20. Davies, P.A. The Latest Developments in Automatic Train Control // Proceedings of the International Conference on Railway Safety Control and Automation Towards 21st Century. London, U.K. 1984. С. 272-279.

21. Turner, D.B., Burns, R.D., Hecht, H. Designing Micro-Based Systems for Fail-Safe Travel // IEEE Spectrum vol. 24, no. 2, 1987. С. 58-63.

22. Уласень А.Ф. Надежность устройств контроля информационных систем управления оружием изделий военного назначения // Научные итоги года: достижения, проекты, гипотезы. 2013. № 3. С. 137-140.

23. Мусин С.М., Дорохов Д.Г., Сутормин Д.А. Надежность программного обеспечения авиационного оборудования летательных аппаратов государственной авиации // Научные чтения по авиации, посвященные памяти Н.Е. Жуковского. 2018. № 6. С. 413-422.

24. Макриденко Л.А., Волков С.Н., Горбунов А.В., Ходненко В.П. История создания, задачи и особенности разработки геостационарного гидрометеорологического космического аппарата "ЭЛЕКТРО" № 1 // Вопросы электромеханики. Труды ВНИИЭМ. 2016. Т. 154. № 5. С. 43-50.

25. Wright, N.C.J. Dissimilar Software // Workshop on Design Diversity in Action. Baden, Austria, 1986.

26. Davis, G.J., Earls, M.R., Patterson-Hine, F.A. Reliability Analysis of the X - 29A Flight Control System Software // Journal of Computer and Software Engineering, vol. 1, no. 4, 1993. С. 325-348.

27. Madden, W.A., Rone, K.Y. Design, Development, Integration: Space Shuttle Primary Flight Software System // Communications of the ACM, vol. 27, no. 8, 1984. С. 902-913.

28. Данилин А.О., Кол М.Д., Петрухнова Г.В. Обеспечение надёжности программных решений // В сборнике: Информатика: проблемы, методология, технологии материалы XV международной научно-методической конференции. 2015. С. 265-268.

29. Прибыткова С.А. Надежность информационно-управляющих систем и их программное обеспечение // Современные тенденции развития науки и технологий. 2015. № 5-2. С. 54-56.

30. Гуров В.В. Практические особенности использования моделей надежности программного обеспечения // Вестник Национального исследовательского ядерного университета МИФИ. 2017. Т. 6. № 5. С. 458-465.

31. Курзакова Н.А. Мультиверсионное программирование как процесс обеспечения надежности программных систем // В сборнике: Информационно-телекоммуникационные системы и технологии материалы Всероссийской научно-практической конференции. 2017. С. 312-314.

32. Викторов Д.С., Пластинина Е.В. Мультиверсионная технология разработки встроенных систем // Вестник Военной академии воздушно-космической обороны. 2016. № 1. С. 65-79.

33. Грузенкин Д.В., Дроздов А.В., Семёнов П.В. Обзор методов реализации мультиверсионного программного обеспечения // Новая наука: От идеи к результату. 2016. № 12-3. С. 53-56.

34. Volochiy, B., Mulyak, O., Ozirkovskyi, L., Kharchenko, V. Automation of quantitative requirements determination to software reliability of safety critical NPP I&C systems // Proceedings - 2nd International Symposium on Stochastic Models in Reliability Engineering, Life Science, and Operations Management, SMRLO 2016 11 March 2016, Номер статьи 7433137, С. 337-346

35. Orges Çiço, Zamir Dika, Betim Çiço, “High reliability approaches in cloud applications for business – reliability as a service (RAAS) model”, International Journal on Information Technologies & Security, № 3, vol. 9, 2017, С. 3-18.

36. Нечаева К.О. Реализация разнообразия при разработке мультиверсионного программного обеспечения // В сборнике: Информационно-телекоммуникационные системы и технологии материалы Всероссийской научно-практической конференции. 2017. С. 321-323.

37. Avizienis, A., Kelly, J.P.J. Fault Tolerance by Design Diversity: Con ceptsand Experiments // IEEE Computer. 1984. С. 67-80.

38. Fairley, R. Software Engineering Concepts. McGraw-Hill, N.Y., 1985.

39. Завьялова О.И., Гриценко С.Н., Тынченко С.В., Царев Р.Ю. Модель формирования оптимальной программной системы по схеме блока восстановления с согласованием // Современные проблемы науки и образования. 2015. № 1-1. С. 297.

40. Соловьев А.Н., Стемпковский А.Л. Методы повышения отказоустойчивости работы устройства управления микросистемы за счет введения структурной избыточности // Информационные технологии. 2014. № 10. С. 17-22.

41. Царев Р.Ю. Алгоритм формирования программной системы по схеме блока восстановления с согласованием на основе нечеткой логики // Современные проблемы науки и образования. 2015. № 2. С. 178.

42. Чувашова Д.А., Ермаков А.А. Возможные методы оценки надежности программного обеспечения // Студенческий, Изд-во: Ассоциация научных

сотрудников "Сибирская академическая книга" (Новосибирск), 2017. № 8-1 (8). С. 36-40.

43. A Standard Classification of Software Errors, Faults and Failures : A Standard for Radix -Independent Floating-point Arith metic. New York : American National Standards Institute/Institute of Electrical and Electronics Engineers, ANSI/IEEE Std. 854-1987. 1987.

44. Комарова Ю.В. Оценка надёжности и безопасности авиационных систем на нечетких множествах // Научный вестник ГосНИИ ГА. 2012. № 2 (313). С. 122-125.

45. Klatte, R., Kulisch, U., Lawo, C., Rauch, M., Wiethoff, A. C-XSC. Springer-Verlag, New York, 1992.

46. Михалев А.С., Исаев А.Н., Носарев К.А. Современные технологии реализации мультиверсионного программного обеспечения // Новая наука: От идеи к результату. 2016. № 12-3. С. 129-133.

47. Рейзмунт Е.М. Разработка инструментальных средств анализа живучести и безопасности оболочечных конструкций технических объектов // Информационные и математические технологии в науке и управлении. 2017. № 2 (6). С. 113-122.

48. Букашкин С.А., Кривошеин Б.Н., Терехов А.Н., Фоминых Н.Ф. Проектирование отказоустойчивого вычислительного комплекса с архитектурой 2 из 3 (2оо3) // Программная инженерия. 2012. № 3. С. 12-20.

49. Бубнов А.А. Основные характеристики программной системы для оценки надежности программного обеспечения // Современные технологии в науке и образовании - СТНО-2017 сборник трудов II Международной научно-технической и научно-методической конференции: в 8 т.. Рязанский государственный радиотехнический университет. 2017. С. 134-135.

50. Feng Chen, Weizhong Qiang, Hai Jin, Deqing Zou, Duoqiang Wang Multi-version Execution for the Dynamic Updating of Cloud Applications // 2015 IEEE

39th Annual Computer Software and Applications Conference, 2015, Volume: 2, C. 185 – 190

51. Avizienis, A., Chen, L. On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution // Proceedings COMPSAC 77. 1977. C. 149-155.

52. Антонов А.В. Особенности анализа надёжности асу тп и её программного обеспечения // В сборнике: Управление развитием крупномасштабных систем MLSD'2016 Труды Девятой международной конференции: в 2-х томах. Под общей редакцией С. Н. Васильева, А. Д. Цвиркуна ; Институт проблем управления им. В.А.Трапезникова РАН. 2016. С. 176-183.

53. Saramud M.V., Zelenkov P.V., Kovalev I.V., Kovalev D.I., Kartsan I.N. Development of methods for equivalent transformation of gert networks for application in multi-version software // IOP Conference Series: Materials Science and Engineering. 12. Series: "XII International Scientific and Research Conference "Topical Issues in Aeronautics and Astronautics"" 2016. C. 012015.

54. Belli, F., Jedrzejowicz, P. Fault-Tolerant Programs and Their Reliability. Transactions Pel., vol. 29(2), 1990. C. 184-192.

55. Scott, R.K., Gault, J.W, McAllister, D.F. Fault-Tolerant Reliability Modeling // IEEE Transactions on Software Engineering, vol. SE-13, no. 5, 1987. C. 582-592.

56. Tso, K.S., Avizienis, A. Community Error Recovery in N-Version Software: A Design Study with Experimentation // Proceedings of the 17th Fault-Tolerant Computing Symposium. 1987. C. 127-133.

57. Tingting Hu, Ivan Cibrario Bertolott, Nicolas Navet Towards seamless integration of N-version programming in model-based design // 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2017, C. 1 – 8

58. Вайтекунене Е.Л. Алгоритм решения задачи мультиверсионного формирования программно-информационных технологий // Актуальные проблемы авиации и космонавтики. 2017. Т. 2. № 13. С. 344-347.

59. Грузенкин Д.В., Новиков О.С., Суханова А.В. Мультиверсонное по и блоки восстановления - два способа защиты от ошибок // Новая наука: От идеи к результату. 2016. № 11-2. С. 72-75.

60. Majid Khabbazian Near-optimal multi-version codes // 2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton), 2015, С. 728 – 732

61. Насыров И.Р., Сташков Д.В. Мультиверсионное программирование в автоматических системах управления технологическими процессами // Приоритетные научные направления: от теории к практике. 2015. № 16. С. 75-78.

62. Stosic, D., Stosic, D., Ludermir, T. Voting based q-generalized extreme learning machine // Neurocomputing Volume 174, 22 January 2016, С. 1021-1030

63. Son, K.S., Kim, D.H., Kim, C.H., Kang, H.G. Study on the systematic approach of Markov modeling for dependability analysis of complex fault-tolerant features with voting logics // Reliability Engineering and System Safety Volume 150, 1 June 2016, С. 44-57

64. Xu, J. The t(n-1)-diagnosability and its applications to fault tolerance // Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium, 1991, С. 496 – 503.

65. Xu, J. Software Fault Tolerance: t/(n-1)-Variant Programming // Jie Xu and Brian Randell // IEEE Transactions on Reliability, Vol. 46, No. 1, March 1997, С. 60 - 68.

66. Липаев В.В. Надежность и функциональная безопасность комплексов программ реального времени // Программная инженерия. 2013. № 8. С. 10-18.

67. Липаев В.В. Управление конфигурацией сложных комплексов программ реального времени // Программная инженерия. 2014. № 2. С. 3-12.

68. Липаев, В. В. Проблемы разработки и обеспечения качества крупномасштабных программных средств / Программирование. 2005. Т. 31. № 1. С. 73-77.

69. Липаев В.В. Основные понятия, факторы и стандарты, определяющие качество крупномасштабных программных средств – М.: Берлин, 2015, 237 с.

70. Липаев, В.В. Анализ и сокращение рисков проектов сложных программных средств. – М.: НПО Синтег, 2005.

71. Липаев, В.В. Методы обеспечение качества крупномасштабных программных средств. – М.: РФФИ. Синтег, 2003.

72. Липаев, В.В. Системное проектирование сложных программных средств для информационных систем. Изд. Второе переработанное и дополненное. – М.: Синтег, 2002.

73. Липаев, В.В. Сопровождение и управление конфигурацией сложных программных средств. – М.: СИНТЕГ, 2006, 372 с.

74. Калинин А.О., Посконин М.В., Сарамуд М.В., Лосев В.В., Ковалев И.В. Методика расчета временных характеристик элементов автоматизированной системы управления на примере замкнутого контура регулирования давления на участке трубопровода под управлением контроллера "ОВЕН ПЛК100 220" // Сибирский журнал науки и технологий. 2017. Т. 18. № 2. С. 387-395.

75. Ковалев И.В. Управление развитием надежных кластерных структур информационных систем / И. В. Ковалев, Н. Н. Джиоева, А. В. Прокопенко, Р. Ю. Царев //Программные продукты и системы. 2010. № 2 (90).С. 68–71.

76. Ковалев, И.В. Модели поддержки многоэтапного анализа надежности программного обеспечения автоматизированных систем управления/ И.В. Ковалев, Р.Ю. Царев, М.А. Русаков, М.Ю. Слободин // Проблемы машиностроения и автоматизации.– № 2, 2005.– С. 73-81.

77. Ковалев, И.В. Мультиверсионный метод повышения программной надежности информационно-телекоммуникационных технологий в

корпоративных структурах / И.В. Ковалев, Р.В. Юнусов; Телекоммуникации и информатизация образования. 2003. №2, С. 50-55.

78. Ковалев, И.В. Надежность архитектуры программного обеспечения телекоммуникационных технологий / И.В. Ковалев, Н.В. Василенко, Р.В. Юнусов; Международная научная конференция Telematica’2001, Санкт-Петербург, 2001– С. 23-24.

79. Kovalev I.V., Zelenkov P.V., Ognerubov S. The practical approach to the reliability analysis of the software architecture of a complex company control system // В сборнике: IOP Conference Series: Materials Science and Engineering Сер. "International Scientific and Research Conference on Topical Issues in Aeronautics and Astronautics (Dedicated to the 55th Anniversary from the Foundation of SibSAU)" 2015. С. 012011.

80. Kovalev I., Losev V., Saramud M., Kuznetsov P., Petrosyan M. To the Question of the Organization of a Learning Environment for Developers of Cross-Platform OnBoard Software for Unmanned Aerial Vehicles // TOJET: The Turkish Online Journal of Educational Technology – December 2017, Special Issue for INTE 2017, С. 700-705.

81. Kovalev I.V., Losev V.V., Saramud M.V., Kovalev D.I., Petrosyan M.O., Brezitskaya V.V. To the question on implementation of multi-version execution environment software of onboard autonomous unmanned objects by means of real-time operating system // Сибирский журнал науки и технологий. 2017. Т. 18. № 4. С. 744-747.

82. Ковалев Д.И., Сарамуд М.В., Карасева М.В., Нургалеева Ю.А. Развитие методов эквивалентного преобразования ГЕРТ-сетей для анализа мультиверсионного программного обеспечения // Вестник Сибирского государственного аэрокосмического университета им. академика М.Ф. Решетнева. 2014. № 1 (53). С. 11-16.

83. Ковалев И.В., Зеленков П.В., Сарамуд М.В., Сидорова Г.А., Брезицкая В.В. Модели ГЕРТ-сетей для различных способов применения методологии

мультиверсий // Вестник Сибирского государственного аэрокосмического университета им. академика М.Ф. Решетнева. 2013. № 1 (47). С. 41-47.

84. Царев, Р.Ю. Метод простого суммарного назначения весов при решении задачи многоатрибутивного выбора состава мультиверсионной программной системы. // Решетневские чтения. Тез. докл. V Всерос. Научн.-практ. конф. студентов, аспирантов молодых специалистов 12-15 ноября 2001. – Красноярск: САА, 2001. – С. 86-87.

85. Chernigovskiy A.S., Tsarev R.Yu., Knyazkov A.N. Hu's algorithm application for task scheduling in N-version software for satellite communications control systems // 2015 International Siberian Conference on Control and Communications (SIBCON), 2015, С. 1 – 4

86. Царев, Р.Ю. Многокритериальное принятие решений при мультиверсионном проектировании гарантоспособных программных средств. // Научная сессия МИФИ – 2002 г. Научн.-тех. конференция «Научно-инновационное сотрудничество». Сборник научных трудов. В 3 частях. – М: МИФИ, 2002. – С. 153-154.

87. Царев, Р.Ю. Многоцелевой выбор проекта информационной системы с эффективным распределением взаимосвязанных ресурсов. // Решетневские чтения. Тез. докл. IV Всерос. Научн.-практ. конф. студентов, аспирантов молодых специалистов 10-12 ноября 2000. – Красноярск: САА, 2000. – С. 273.

88. Царев, Р.Ю. Модель многокритериального решения при выборе проектов мультиверсионной программной системы. / Р.Ю. Царев, Ю.Г. Шиповалов // Перспективные материалы, технологии, конструкции, экономика. Материалы Всероссийской научно-технической конференции 24-26 мая 2001. – Вып.7. – Красноярск: ГАЦМиЗ, 2001. – С. 642-644.

89. Царев Р.Ю. Применение сетей петри при моделировании программ с блоком восстановления /Царев Р.Ю., Тынченко С.В., Гриценко С.Н. // Современные наукоемкие технологии. 2016. № 6-2. С. 310-314.

90. Царев Р.Ю. Оценка транзакционной надежности современных систем управления и обработки информации / Р. Ю. Царев, А. В. Штарик, Е. Н. Штарик, О. И. Завьялова // Приборы и системы. Управление, контроль, диагностика. 2012. № 6. С. 29–32.

91. Царев Р.Ю. Анализ архитектурной надежности программного обеспечения информационно-управляющих систем / Ковалев И.В., Царев Р.Ю., Завьялова О.И. // Приборы. 2010. № 11. С. 24-26.

92. Царев Р.Ю. Формирование высоконадежных программных средств телекоммуникационных систем реального времени // Современные наукоемкие технологии. 2010. № 9. С. 119-122.

93. Царев Р.Ю. Методы повышения надежности программного обеспечения / Русаков М.А., Царев Р.Ю. // В мире научных открытий. 2011. № 8. С. 32-42.

94. Ярославский Д.А., Иванов Д.А., Горячев М.П., Гайнутдинов А.Р., Садыков М.Ф. Выбор операционной системы реального времени для беспроводного устройства // Вестник Казанского государственного технического университета им. А.Н. Туполева. 2016. Т. 72. № 4. С. 95-100.

95. Курниц, А. FreeRTOS —операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2 (115). С. 96-100.

96. Курниц, А. FreeRTOS —операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 3 (116). С. 109-114.

97. Курниц, А. FreeRTOS —операционная система для микроконтроллеров // Компоненты и технологии. 2011.№ 7 (120). С. 23-32.

98. Курниц, А. FreeRTOS —операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 10 (123). С. 93-100.

99. Курниц, А. FreeRTOS —операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 11 (124). С. 99-108.

100. Курниц А. FreeRTOS. Взгляд изнутри. алгоритм работы планировщика. Часть 1 // Компоненты и технологии. 2013. № 5 (142). С. 114-122.

101. Курниц А. FreeRTOS. Взгляд изнутри. алгоритм работы планировщика. Часть 2 // Компоненты и технологии. 2013. № 6 (143). С. 89-94.

102. Юров А.Н., Рыжков В.А., Паринов М.В. Использование программных интерфейсов API для разработки подсистем САПР // Информатика: проблемы, методология, технологии материалы XV международной научно-методической конференции. 2015. С. 398-403.

103. Исаев А., Акиншин Л. Система реального времени или Windows? // Windows IT Pro/ RE. 2009. № 12. С. 70-74.

104. Laprie, J.-C., Arlat, J., Beounes, C., and Kanoun, K. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures // IEEE Computer, vol. 23, no. 7, July 1990, pp. 39-51. Reprinted in Fault-Tolerant Software Systems: Techniques and Applications, Hoang Pham (ed.), IEEE Computer Society Press, 1992, C. 5-17.

105. Ковалев, И.В. Анализ проблем в области исследования надежности программного обеспечения: многоэтапность и архитектурный аспект / И.В. Ковалев // Вестник СибГАУ. Выпуск № 3 (55), Красноярск, 2012 с. 78-92.

106. Kovalev I., Losev V., Saramud M., Petrosyan M. Model implementation of the simulation environment of voting algorithms, as a dynamic system for increasing the reliability of the control complex of autonomous unmanned objects // (2017) MATEC Web of Conferences, 132, статья № 04011.

107. Kovalev I., Voroshilova A., Losev V., Saramud M., Chuvashova M., Medvedev A. Comparative Tests of Decision Making Algorithms for a Multiversion Execution Environment of the Fault Tolerance Software // European Conference on Electrical Engineering and Computer Science (EECS 2017).

108. Азымшин И.М., Чуканов В.О. Определение надёжности программного обеспечения с учётом возможности появления новых ошибок при исправлении ранее обнаруженных // Безопасность информационных технологий. 2015. № 3. С. 6-9.

109. Kharchenko V., Fesenko H., Sachenko A., Hiromoto R.E., Kochan V. Reliability issues for a multi-version post-severe NPP accident monitoring system // 2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2017, Volume: 2, C. 942 – 946

110. Xiaogang Song, Zhengjun Zhai, Ruoning Lv, Yangming Guo, Peican Zhu Optimal Design of the k -Out-of- n : G (F) Majority Voter // IEEE Access, 2017, Volume: 5, C. 22647 - 22656

111. P. Lakshmi Priya, N. Kirubanandasarathy A novel fault tolerant voter for identical core in satellite application // 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), 2016, C. 814 - 818

112. Algirdas Avizienis, The Methodology of N-Version Programming, in R. Lyu, ed itor // Software Fault Tolerance, John Wiley & Sons, 1995.

113. Avizienis, A. The N-Version Approach to Fault -Tolerant Software // IEEE Transactions on Software Engineering, vol. SE-11, no. 12, 1985. C. 1491-1501.

114. Juraj Slačka; Miroslav Halás Safety critical RTOS for space satellites // 20th International Conference on Process Control (PC), 2015, C. 250 – 254

115. Nassaj, A., Barabady, J. Fault tree analysis of oil and gas distillation tower and application of Bayesian Networks // IEEE International Conference on Industrial Engineering and Engineering Management Volume 2016-December, 27 December 2016, Номер статьи 7798161, C. 1669-1673

116. Broen, R.B. New Voters for Redundant Systems, Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control, March 1975, pp. 41 – 45.

117. Сарамуд М.В., Зеленков П.В., Брезицкая В.В., Ковалев И.В., Ковалев Д.И., К вопросу надежности программного обеспечения отказоустойчивых систем // Материалы Международной научно-практическая конференции «Современное состояние науки и техники», 2016 г. С. 149-151.

118. Peng Jiao, Kai Xu, Lin Sun Diversity voting and its application in real-time strategy games multi-objective optimization decision-making behavior modeling // 2016 3rd International Conference on Systems and Informatics (ICSAI), 2016, C. 552 - 559

119. Shih-Chun Chou, Yuan-Hao Chang, Yuan-Hung Kua, Po-Chun Huang, Che-Wei Tsao Multi-version checkpointing for flash file systems // 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), 2016, C. 436 – 443

120. Randell, B. The Evolution of the Recovery Block Concept / J. Xu, B. Randell – University of Newcastle upon Tyne, England, 1995.

121. Filip Veljković, Teresa Riesgo, Eduardo de la Torre Adaptive reconfigurable voting for enhanced reliability in medium-grained fault tolerant architectures // 2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2015, C. 1 – 8

122. Kim, K. Fault-Tolerant Software Voters Based on Fuzzy Equivalence Relations / K.Kim, M.A. Vouk and D.F. McAllister – Department of Computer Science, North Carolina State University, 1997

123. Котенок, А.В. Реализация алгоритмов мультиверсионного голосования / А.В. Котенок // Вестник университетского комплекса: Сб.научн. трудов / Под общей ред. профессора Н.В. Василенко; Красноярск: ВСФ РГУИТП, НИИ СУВПТ.- 2004. Выпуск 3 (17). - с. 86-93.

124. Лебедев В.В., Чернышев О.Л. Тестирование надёжности программного обеспечения системы автоматического управления // В сборнике: Информационные ресурсы и системы в экономике, науке и образовании Сборник статей VII Международной научно-практической конференции. 2017. С. 55-60.

125. Поздняков, Д.А. Разработка и исследование среды мультиверсионного исполнения программных модулей. / Д.А. Поздняков, И.С. Титовский, Р.В.Юнусов // Вестник НИИ СУВПТ: Сб. научн. трудов; Красноярск: НИИ СУВПТ.- 2003. Выпуск 13. С. 155-170

126. Kovalev I.V., Zelenkov P.V., Losev V.V., Kovalev D.I., Ivleva N.V., Saramud M.V. Multiversion environment creation for control algorithm execution by autonomous unmanned objects // IOP Conference Series: Materials Science and Engineering. 5. Series: "V International Workshop on Mathematical Models and their Applications 2016" 2017. С. 012025.

127. Аксененко И.А. Объектно ориентированный подход к разработке мультиверсионного программного обеспечения // Решетневские чтения. 2015. Т. 2. № 19. С. 6-8.

128. Jing Liu, Na Yang Optimal fault tolerant service provisioning for cloud application // 2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC), 2017, С. 189 – 194

129. Ковалев, И.В. К проблеме выбора алгоритма принятия решения в мультиверсионных системах / И.В. Ковалев, А.В. Котенок // Информационные технологии; №9, 2006. с. 39-44.

130. Avizienis, A. The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software // Digest of Papers: The Fifteenth Annual International Symposium on Fault-Tolerant Computing (FTCS 15), Ann Arbor, Michigan, June 19 – 21, 1985, pp. 126 – 134.

131. Павленко Е.Г., Калобутин В.К. Среда исполнения мультиверсионного программного обеспечения, вид изнутри // В сборнике: Проблемы эффективности и безопасности функционирования сложных технических и информационных систем, сборник статей по итогам Международной научно-практической конференции, 2017, С. 82-85.

132. Сарамуд М.В., Зеленков П.В., Ковалев И.В., Ковалев Д.И., Брезицкая В.В., Характеристика надежности программных модулей // Решетневские чтения. 2015. Т. 2. № 19. С. 87-88.

133. Машкин А.В. Технологии разработки программного обеспечения // Учебное пособие / Вологда, 2014, 75 с.

134. Avizienis, A. Dependable Computing Depends on Structured Fault Tolerance, Proceedings of the 1995 6t h International Symposium on Software Reliability Engineering, Toulouse, France, 1995, C. 158 – 168.

135. Xu, J. Implementing Software-Fault Tolerance in C++ and Open C++: An Object-Oriented and Reflective Approach / J. Xu, B. Randell, A.F. Zorzo – Department of Computing Science, University of Newcastle upon Tyne, 2000

136. Григорьев И.В., Шершуков М.С., Яхин Р.М., Давлетшин Р.С. Электронный учебник по методам оптимизации // Хроники объединенного фонда электронных ресурсов Наука и образование. 2013. № 12 (55). С. 90.

137. Кузина В.В., Кошев А.Н. Численные методы и методы оптимизации // Лабораторный практикум для направления подготовки 09.03.02 "Информационные системы и технологии" / Пенза, 2016, 104 с.

138. Сербулов Ю.С. Методы решения задач оптимизации. основные методы теории оптимизации // лабораторный практикум / Ю. С. Сербулов. Воронеж, 2016, 96 с.

139. Тананко И.Е., Долгов В.И. Сборник задач по методам оптимизации // Учебно-методическое пособие / Саратов, 2017, 32 с.

140. [Электронный ресурс]. Операционные системы реального времени для начинающих. Режим доступа (URL: https://habrahabr.ru/post/208780/).

141. Шангареева Г.Р., Мустафина С.А. О численном методе решения задач оптимизации // В сборнике: СОВРЕМЕННАЯ МАТЕМАТИКА И ЕЕ ПРИЛОЖЕНИЯ материалы международной научно-практической конференции. 2017. С. 400-403.

142. Saramud M.V., Kovalev I.V., Losev V.V., Kuznetsov P.A. Software interfaces and decision block for the execution environment of multi-version software in real-time operating systems // International Journal on Information Technologies & Security, No 1 (vol. 10), 2018, pp. 25-34.

143. Козленко Л. Проектирование информационных систем. // М.: КомпьютерПресс, № 9-11, 2001.

144. Ivan Cibrario Bertolotti RTOS support in C-language toolchains // IEEE International Conference on Industrial Technology (ICIT), 2017, С. 1328 - 1333

145. Страуструп, Б. Дизайн и эволюция C++. / Б. Страуструп – СПб.: Питер, 2006. – 448 с.

146. Карпов А.В. Аспекты применения предметно-ориентированного подхода к проектированию сложных программных систем // Программные продукты и системы. 2013. № 2. С. 16.

147. Marco Gaudesi, Irith Pomeranz, Matteo Sonza Reorda, Giovanni Squillero New Techniques to Reduce the Execution Time of Functional Test Programs // IEEE Transactions on Computers, 2017, Volume: 66, Issue: 7, С. 1268 - 1273

148. Острейковский, В.А. Теория систем / В.А. Острейковский // М.: Высш. шк., 1997. 239с.

149. Горохов А.В. Основы системного анализа // Учебное пособие / Москва, 2017. Сер. 11 Университеты России (1-е изд.)

150. Цилюрик, О. И. QNX/UNIX: Анатомия параллелизма // – М.: Символ, 2011, 308с.

151. Царев Р.Ю., Капулин Д.В., Машурова Д.В., Тынченко Я.А., Ковтанюк Д.Н. Многоатрибутивное формирование гарантоспособных систем управления и обработки информации // Вестник Сибирского государственного аэрокосмического университета им. академика М.Ф. Решетнева. 2012. № 5 (45). С. 106-110.

152. ГОСТ 27.002-89. Надежность в технике. Основные понятия. Термины и определения" (утв. Постановлением Госстандарта СССР от 15.11.1989 N 3375)

153. ГОСТ 28195-99 «Оценка качества программных средств» - Межгосударственный Совет по стандартизации, метрологии и сертификации, Минск.