УДК 004.41(075.8)

Вестник СибГАУ
2014. № 3(55). С. 78–92

АНАЛИЗ ПРОБЛЕМ В ОБЛАСТИ ИССЛЕДОВАНИЯ НАДЕЖНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ: МНОГОЭТАПНОСТЬ И АРХИТЕКТУРНЫЙ АСПЕКТ

И. В. Ковалев

Сибирский государственный аэрокосмический университет имени академика М. Ф. Решетнева
Российская Федерация, 660014, г. Красноярск, просп. им. газ. «Красноярский рабочий», 31
E-mail: kovalev.fsu@mail.ru

В рамках обзора показано, что оценка и анализ надежности программного обеспечения (ПО) должны осуществляться на каждом шаге создания отказоустойчивых программ. В любом случае очень важно использовать как можно больше априорной информации перед началом каждой фазы создания ПО, так как это обеспечит наиболее приемлемые и точные результаты. Так как нет единого подхода, методики и параметров для оценки надежности ПО, в настоящее время существует множество моделей и алгоритмов, проверенных на практике, но имеющих ряд недостатков. Проводится анализ методов, моделей и методологий в области создания надежного, отказоустойчивого ПО. Делается попытка систематизировать и унифицировать существующие методологии. Учитывается многоэтапность создания ПО и архитектурный аспект в рамках всего жизненного цикла отказоустойчивого программного обеспечения.

Ключевые слова: программное обеспечение, надежность, программная архитектура, этап разработки, отказоустойчивость, жизненный цикл.

Vestnik SibGAU
2014, No. 3(55), P. 78–92

ANALYSIS OF PROBLEMS IN THE RESEARCH AREA OF SOFTWARE RELIABILITY: A LOT OF STAGES AND ARCHITECTURAL ASPECT

I. V. Kovalev

Siberian State Aerospace University named after academician M. F. Reshetnev
31, Krasnoyarsky Rabochy Av., Krasnoyarsk, 660014, Russia
E-mail: kovalev.fsu@mail.ru

In the survey, it is shown that estimation and reliability analysis should be performed at each step, create a failover. Due to the fact that there is no single approach, methods and parameters for estimating the reliability of Software*, there are many models and algorithms, tested virtually, but has several disadvantages. The main drawback of all models is "specialization". Models are always tied to any one phase of software development, and sometimes have a purely theoretical value due to the fact that they must be used the data obtained at other stages of development, and the authors of such models usually consider this fact of secondary importance.*

The most promising approach is aimed at creating some hybrid model, which includes all the advantages of a well-validated models and algorithms. Existing models and algorithms need to be modified so that the output of one model became the input for the other; to provide a unifying parameters, units of measure conversion, normalization of data and satisfying the current requirements and trends, such as object-oriented analysis and programming. From this review we can see that it is very important to use as much a priori information before beginning each phase of creation, as this will provide the most appropriate and accurate results.

The analysis of methods, models and methodologies in the field of creation of reliable, fault-tolerant (Software) is carried out. Attempts to systematize and unify the existing methodology are made. The multiple steps involved in the creation and architectural aspect within the entire life-cycle fault-tolerant software are taken into account.

Keywords: Software reliability, software architecture, development stage, resiliency, life cycle.

Интерес к оцениванию надежности программного обеспечения (ПО) возник одновременно с появлением программ. Он был вызван естественным стремлением получить традиционную вероятностную оценку надежности технического устройства (ЭВМ или персонального компьютера), работа которого в основном и предназначалась для функционирования ПО. Последнее было определено как одна из составляющих

частей машины, поэтому подход к оцениванию надежности программной части первоначально мало отличался от оценивания надежности техники и заключался в переносе известных статистических методов классической теории надежности на новую почву, образовав ее отдельную ветвь – теорию надежности ПО [1–3]. В целом этот подход сохранился до сегодняшнего дня. Однако по мере развития вычислительной техники пришло четкое понимание того, что ПО не просто составная часть вычислительной техники. В современных условиях развития цифровой техники специализированное ПО перестало быть принадлежностью одной вычислительной системы (как это было раньше), а стало использоваться на сотнях и тысячах аналогичных компьютеров (в основном персональных). Даже если не касаться вопросов информационной безопасности, проблема обеспечения устойчивого функционирования расчетных программ, выявления их ошибок сегодня крайне остро стоит перед разработчиками [4].

За прошедшие десятилетия было создано множество методов, методик и моделей исследования надежности ПО. Однако, к сожалению, единого подхода к решению этой проблемы предложено не было и, по-видимому, в ближайшее время не предвидится. Тем не менее при разработке сложных программных систем их создатели стараются в той или иной степени получить оценку надежности ПО. Более правильный подход заключается в последовательном оценивании надежности программ на каждом этапе разработки. Основная сложность при использовании статистических методов заключается в отсутствии достаточного количества исходных данных. Динамика выявления ошибок должна тщательно фиксироваться и обрабатываться. Важной проблемой является степень детализации элемента расчета надежности. Выявить все связи обработки информации (как это порой предлагается) даже для достаточно несложной программы практически нереально. Исходя из этого, детализация элементов расчета надежности (условно называемых программными модулями) должна ограничиваться законченными программными образованиями, которые, взаимодействуя между собой, составляют более сложное объединение (комплекс), надежность которого нас интересует. При этом допускается, что надежность самой вычислительной техники, операционной системы (ОС) и среды программирования полная, нас интересует лишь надежность функционирования специальных программных средств, решающих основную целевую задачу системы.

В данной работе проводится анализ методов, моделей и методологий в области создания надежного, отказоустойчивого ПО; делается попытка систематизировать и унифицировать существующие методологии; учитывается многоэтапность создания ПО и архитектурный аспект в рамках всего жизненного цикла гарантоспособного программного обеспечения.

В результате анализа многих работ из области исследования надежности ПО был выявлен широкий ряд проблем, основной причиной которого является отсутствие единого стандарта в области отказоустойчивого ПО. Это показывает анализ многих работ как зарубежных исследователей [5–15], так и отечественных авторов [1; 2; 16–26]. Среди множества различных институтов и организаций (ISREE, IEEE, NASA), независимых исследователей проблем надежности ПО не существует единой терминологии, единых параметров и показателей надежности программ, а также методологии разработки отказоустойчивого ПО [3; 27–48].

Можно выделить три основные группы проблем в данной области:

– отсутствие единой методологии создания отказоустойчивого ПО;

– отсутствие единой методологии тестирования отказоустойчивого ПО;

– отсутствие единого подхода к анализу проблемной области.

Касаясь проблем терминологии, отметим недостатки существующих определений, когда программное обеспечение содержит ошибку [49]:

1. Поведение ПО не соответствует спецификациям.

Недостатки: неявно предполагается, что спецификации корректны. Подготовка спецификаций – один из основных источников ошибок. Если поведение программного продукта не соответствует его спецификациям, ошибка, вероятно, имеется. Однако если система ведёт себя в соответствии со спецификациями, мы не можем утверждать, что она не содержит ошибок.

2. Поведение ПО не соответствует спецификациям при использовании в установленных при разработке пределах.

Недостатки: если система случайно используется в непредусмотренной ситуации, её поведение должно оставаться разумным. Если это не так, она содержит ошибку.

3. Программное обеспечение ведёт себя не в соответствии с официальной документацией и поставленными пользователю спецификациями.

Недостатки: ошибки могут содержаться и в программе, и в спецификациях, или в руководстве описана только ожидаемая и планируемая работа с системой.

4. Система не способна действовать в соответствии с исходной спецификацией и перечнем требований пользователя.

Это утверждение тоже не лишено недостатков, поскольку письменные требования пользователя редко детализированы настолько, чтобы описывать желаемое поведение программного обеспечения при всех мыслимых обстоятельствах.

Окончательное определение: в программном обеспечении имеется ошибка, если оно не выполняет того, что пользователю разумно от него ожидать. Отказ программного обеспечения – это проявление ошибки в нём.

Термины «ошибка», «сбой» и «дефект» часто используются без разделения по смыслу. В ПО «ошибка» – это действия программиста, которые

приводят к дефектам в ПО. «Дефект» в ПО влечет за собой сбои во время исполнения кода [3]. «Сбой» – отклонение выхода системы от желаемого при выполнении кода.

Дефект влечет за собой сбой только тогда, когда выполняется код, содержащий ошибку, и распространяется до выхода системы. Уровень тестируемости дефекта определяется как вероятность обнаружения сбоя на случайно выбранном выходе.

Уровни сбоя: катастрофичный, высокий, средний, низкий, незначительный. Определения этих уровней меняются от системы к системе.

Простой или «зависание» – особый вид сбоя, зависящий от сбоя как в аппаратной части, так и в программной части системы или же от некорректных действий пользователя.

Организация IEEE определяет надежность как «способность системы или компонента выполнять требуемые функции при определенных условиях за определенный период времени». Это определение дал А. Н. Авиженис, и оно считается классическим [5]. Основной недостаток такого определения – это то, что в нем не учтено различие между ошибками разных типов. Надежность ПО определяется как вероятность функционирования без ошибок за определенный период времени в определенной операционной системе. Для большинства разработчиков надежность определяется как корректность ПО, т. е. количество ошибок, которое надо исправить во время теста.

Существует и другое определение надежности – операционная (транзакционная) надежность. Операционная надежность – это способность системы или компонента выполнять требуемые функции при определенных условиях в рамках транзакции. Широко используется в системах обработки и хранения информации в базах данных [22; 26]. Далее будем использовать понятия классической и транзакционной надежности в зависимости от места их применения и используемой модели надежности.

Следует отметить, что при оценке надежности ПО существуют и другие определения, имеющие немаловажное значение. Одним из таких параметров является время. Время можно разделить на два типа: календарное время и процессорное время. Процессорное время преобразуется в календарное для сравнения параметров ПО, используемого на разном аппаратном обеспечении и в разных операционных системах. При комбинировании различных моделей оценки надежности следует учитывать тот факт, что понятия времени в них могут различаться. Такие модели надо унифицировать, приводя их единому пониманию времени, вводя соответствующие формулы с коэффициентами пересчета времени.

Среди множества работ из области оценки параметров надежности ПО можно выявить несколько десятков подходов к измерению или методам оценки тех или иных количественных показателей, характеризующих надежность [50]. Отметим основные из них [51–53].

Надежность – имеет обозначение R (reliability) и измеряется как вероятность невозникновения сбоя. Надежность используется практически во всех моделях как основной показатель.

Среднее время появления сбоя – имеет аббревиатуру MTTF (Mean Time To Failure) и измеряет время между двумя последовательными сбоями.

Интенсивность сбоев – величина, обратная MTTF, определяет количество сбоев в единицу времени.

Среднее время простоя (TR) – величина, определяющая время, затрачиваемое на выявление, устранение и восстановление системы или компонента системы после сбоя.

Коэффициент готовности системы (S) – определяется как отношение разности среднего времени появления сбоя и среднего времени простоя системы к среднему времени появления сбоя.

Количество оставшихся ошибок в коде ПО – используется при разработке ПО. Показывает количество ошибок в коде на каждую тысячу строк исходного кода.

Плотность ошибок в коде – обычно измеряется как количество ошибок на тысячу строк исходного кода [37].

Представим следующую классификацию существующих моделей, методов и алгоритмов оценки надежности программных средств.

Классификация по оптимизируемым параметрам надежности. К этому классу относятся модели оптимизации сроков и стоимости проекта. Такие модели используются для минимизации сроков разработки и стоимости проекта при сохранении должного уровня надежности ПО. Как правило, используются на ранних стадиях жизненного цикла разработки ПО [33; 40; 46].

Модели выбора оптимального набора мультиверсий в *N*-версионном ПО используются разработчиками для поиска множества недоминируемых решений, удовлетворяющих критерию максимума надежности при минимальных затратах [12; 20–24; 28; 36–38].

Классификация по используемым формальным методам. По формальным методам модели делятся следующим образом:

1. Формальные математические модели – используют какую-либо математическую функцию для экстраполяции параметров надежности, например, для оценки плотности ошибок в коде используют экспоненциальную функцию.

2. Модели поведения ПО – делятся на статистические модели и модели ветвления хода выполнения (path-based) [34].

3. Модели ветвления – используют описание внутренней структуры архитектуры ПО для оценки надежности, при этом описываются все возможные пути выполнения кода и оценивается надежность каждого пути.

4. Статистические модели, использующие исторические данные – аналогично формальным математическим моделям используют статистическую

функцию для экстраполяции параметров надежности. Самые распространенные модели – это модели, использующие регрессию, с помощью которых по накопленным статистическим данным можно достаточно точно оценивать параметры надежности.

5. Марковские цепи – используют для описания поведения структуры программного обеспечения, в таких моделях надежность ПО рассматривается как матрица вероятностей сбоев компонент архитектуры [23; 24; 34].

6. Нейросети – используются для экстраполяции параметров надежности. Для обучения нейросетей используют исторические данные или выходные данные других моделей [54].

Классы моделей по их месту в процессе разработки можно представить следующим образом [55–60]:

– модели, использующиеся на стадии анализа;

– модели, использующиеся на стадии дизайна архитектуры;

– модели, использующиеся на стадии создания кода;

– модели, использующиеся на стадии тестирования;

– модели, использующиеся на стадии опытной эксплуатации;

– модели, использующиеся на стадии промышленной эксплуатации.

Известен также ряд моделей, использующих информацию об ошибках в коде, которые используются для предсказания количества ошибок в коде и, следовательно, зависимости надежности от количества строк в коде, оценки времени исправления оставшихся ошибок в коде. Модели, использующие информацию о внутренней структуре ПО, вычисляют оценку надежности как ПО, состоящего из компонент, или используются для моделирования структуры ПО как множества путей ветвления хода выполнения программы. Модели, использующие искусственную эмуляцию ошибок, оценивают надежность путем прямого теста ПО с искусственным внедрением ошибочных данных или эмуляцией сбоев в аппаратном обеспечении.

Отдельный класс моделей – это гибридные модели. Они используют различные комбинации из всех вышеперечисленных классов моделей. Могут использоваться для оценки любых параметров надежности на разных этапах жизненного цикла программного обеспечения.

Методы и средства обеспечения надежности программного обеспечения. Методы проектирования надежного программного обеспечения можно разбить на следующие группы:

1) предупреждение ошибок – методы, позволяющие минимизировать или исключить появление ошибки;

2) обнаружение ошибок – методы, направленные на разработку дополнительных функций программного обеспечения, помогающих выявить ошибки;

3) устойчивость к ошибкам – дополнительные функции программного обеспечения, предназначенные для исправления ошибок и их последствий, обеспечивающие функционирование системы при наличии ошибок.

Методы предупреждения ошибок концентрируются на отдельных этапах процесса проектирования программного обеспечения и включают в себя:

– методы достижения большей точности при преобразовании информации;

– методы улучшения обмена информацией;

– методы немедленного обнаружения и устранения ошибок.

Методы обнаружения ошибок базируются на введении в программное обеспечение системы различных видов избыточности:

1. Временная избыточность. Использование части производительности аппаратных средств для контроля исполнения и восстановления работоспособности ПО после сбоя.

2. Информационная избыточность. Дублирование части данных информационной системы для обеспечения надёжности и контроля достоверности данных.

3. Программная избыточность. Включает в себя: «взаимное недоверие» – компоненты системы проектируются, исходя из предположения, что другие компоненты и исходные данные содержат ошибки, и должны пытаться их обнаружить; немедленное обнаружение и регистрацию ошибок; выполнение одинаковых функций разными модулями системы и сопоставление результатов обработки; контроль и восстановление данных с использованием других видов избыточности.

Методы обеспечения устойчивости к ошибкам направлены на минимизацию ущерба, вызванного появлением ошибок, и включают в себя:

– обработку сбоев аппаратуры;

– повторное выполнение операций;

– динамическое изменение конфигурации;

– сокращенное обслуживание в случае отказа отдельных функций системы;

– копирование и восстановление данных;

– изоляцию ошибок.

Рассмотрим распределение моделей и методов оценки параметров надежности по фазам жизненного цикла разработки программных средств. В общем случае жизненный цикл ПО можно разделить на следующие фазы [1–3; 16–19].

1. Концептуальная фаза. Построение общей концепции будущей информационно-управляющей системы. Аналитики и эксперты проблемной области изучают поставленные задачи. Результатом фазы является общая концепция построения системы.

2. Определение требований. На данном этапе разработчики взаимодействуют с заказчиком для выявления потребностей и требований к системе. На этом этапе делается незначительное количество ошибок. Изменение требований на более поздних стадиях может повлечь появление большого количества ошибок.

3. Дизайн архитектуры. Строится спецификация взаимосвязей модулей таким образом, что каждый модуль четко разделен, разрабатывается и тестируется независимо. Дизайн может пересматриваться для выявления ошибок.

4. Кодирование. На данном этапе пишется реальный код программы, как правило, на языке высокого уровня. Иногда возможно использование низкоуровневых языков программирования для достижения высокой производительности или для сопряжения с нестандартной аппаратурой. Код постоянно анализируется на предмет наличия ошибок.

5. Тестирование. Эта фаза одна из самых важных и трудоемких. Может занимать от 30 до 60 % временных и материальных ресурсов проекта. Обычно разделяется на несколько последовательных фаз:

5.1. Тестирование модулей. На этом этапе каждый модуль тестируется отдельно, делаются модификации модуля для исправления ошибок. Каждый модуль значительно меньше всей программы и может быть более тщательно протестирован.

5.2. Групповое тестирование. Модули группируются на подсистемы. Подсистемы тестируются независимо. Позволяет выявить ошибки в интерфейсах между модулями. Добавление и удаление модулей из подсистемы позволяет проще идентифицировать ошибки.

5.3. Тестирование системы. Тестируется вся система полностью. Отладка происходит до тех пор, пока не будет достигнут какой-либо критерий. Цель данного этапа – определить максимально возможное количество ошибок за минимальное время.

5.4. Тестирование приемлемости. Цель – оценка производительности и надежности в определенной операционной системе. Собирается информация о том, как пользователи будут использовать систему. Это называется альфа-тестированием. Могут привлекаться реальные пользователи – в этом случае способ называется бета-тестированием.

6. Использование. Система эксплуатируется, и ошибки, выявленные пользователями, не исправляются, пока не выйдет новая версия ПО.

7. Регрессивное тестирование. Новая версия тестируется на работоспособность и проверяется, не уменьшилась ли (регрессировала) надежность ПО.

Наиболее важными являются фазы, которые вносят наибольший вклад в надежность ПО: фаза дизайна архитектуры, фаза кодирования, фаза тестирования компонент и фаза тестирования системы.

Фаза дизайна архитектуры программного обеспечения. Широкое применение средств вычислительной техники породило понятие «архитектура ПО», которое, несмотря на свою распространенность, воспринимается, как правило, интуитивно и употребляется чаще всего при сравнении вычислительных машин.

Под архитектурой ПО, как и вообще любых других средств обработки информации, понимают в узком смысле совокупность их свойств и характеристик, призванных удовлетворить запросы пользователей. В более широком смысле архитектура ПО – это обобщенная модель информационной системы, достаточная для понимания принципов ее функционирования.

Разработка архитектуры – это процесс разбиения большой системы на более мелкие части. Для обозначения этих частей придумано множество названий: программы, компоненты, подсистемы и уровни абстракции. Процесс разработки архитектуры – необходимый шаг в процессе проектирования ПО или комплекса программ (КП). КП представляет собой набор решений множества различных, но связанных между собой задач.

Надежность архитектуры включает в себя надежность индивидуальных элементов. Сбой отдельного элемента приводит к неработоспособности этого и возможно других элементов, но не всей системы в целом. Сбой в системе или ее функциях может выразиться в периоде простоя системы. Период простоя системы определятся как отрезок времени, на котором система или ее часть не может выполнять функции. Период простоя системы приводит к резкому снижению производительности, поэтому уменьшение времени простоя системы является одним из наиболее важных факторов при разработке архитектур распределенного программного обеспечения.

Основные понятия теории надежности комплексов программ базируются на понятиях теории надежности, первоначально развивавшейся применительно к аппаратным комплексам, однако имеются существенные различия в принципах обеспечения надежности программного обеспечения и других технических систем.

В рамках определенного программного проекта трудоемкость создания и возможность достижения заданных параметров надежности определяются, с одной стороны, объемами выделенных ресурсов, а с другой – технологическими средствами, используемыми при проектировании ПО [60].

Для достижения низких показателей сбоев для компонентов аппаратного обеспечения можно увеличить надежность дублированием наиболее критических компонентов архитектуры. В некоторых случаях при отказе компонента его функции могут выполняться дублирующим компонентом. Надежность программного обеспечения нельзя увеличить таким методом, поскольку дублирование компонентов ПО приведет также к дублированию его ошибок. Существуют методы, использующие разнообразие в спецификации, разработке, внедрении и тестировании.

Эксперименты показывают, что применение избыточности на самых ранних этапах разработки, насколько это возможно, уменьшает вероятность появления ошибок. Надежное ПО можно создать с помощью тщательной разработки архитектуры и выявления ошибок в компонентах, которые больше всего оказывают влияние на надежность системы. Эти компоненты определяются как наиболее часто

используемые или архитектурно связанные с множеством других компонентов, влияя, таким образом, на их надежность.

Модульная организация предполагает деление ПО на функционально завершенные части (модули), унификацию связей между ними и установление иерархии взаимодействия компонентов, которая определяется последовательностью их вызова. Вызов осуществляется передачей управления от вызывающего модуля к вызываемому, который по окончании выполнения возвращает управление вызывающему модулю. Вызов группы программ осуществляется передачей управления диспетчеру группы программ более низкого уровня иерархии, входящей в данный КП.

Основные методы современной практической разработки программных комплексов базируются на функциональной декомпозиции с использованием модульно-иерархических принципов [26]. При этом на каждом иерархическом уровне ограничивается сложность компонентов и их связей. В результате общая сложность системы растет значительно медленнее с возрастанием объема задач, чем при неструктурированном проектировании. Эти принципы привели к созданию структурного программирования как стандартного способа построения модульных программ.

Определение модульной программы опирается на ограничения по размерам программ и на понятие их независимости, т. е. модульная программа должна состоять из модулей конечных размеров, которые имеют точку входа и точку выхода. Преимущества модульности состоят в упрощении проектирования и модификации программы, облегчении тестирования и отладки программ, возможности создания библиотеки стандартных модулей и т. д. Недостатки – увеличение времени исполнения программ и объема памяти, усложнение межмодульного взаимодействия – в целом окупаются выигрышем, получаемым от сокращения срока разработки и упрощения сопровождения программ.

Основным средством реализации модульности программ является структурное программирование. Целями структурного программирования служат повышение читабельности и ясности программ, увеличение производительности работы программистов и упрощение процесса разработки. Само понятие структурного программирования представляет собой некоторые принципы написания программ в соответствии с совокупностью определенных правил. Согласно «структурной теореме» для построения любой программы необходимы три основные базовые конструкции:

1) простая вычислительная последовательность, означающая, что два действия должны выполняться одно за другим;

2) альтернатива или ветвление, при котором на основе проверки некоторого условия делается выбор между двумя возможными путями;

3) цикл или итерация, обеспечивающая повторное выполнение некоторой последовательности действий.

Применяемый при структурном программировании сквозной структурный контроль необходим для обнаружения и исправления ошибок на ранних стадиях проектирования, пока стоимость исправления ошибок минимальна, а последствия их наличия незначительны. Такой контроль обеспечивается регулярным обсуждением принятых решений всеми взаимодействующими разработчиками. Кроме того, базовые структуры, унификация правил взаимодействия компонентов и заведомо известная последовательность их разработки в порядке соподчиненности создают предпосылки для автоматизации процесса структурного контроля.

Модульность построения приводит к использованию иерархической структуры взаимодействия модулей программы. Иерархическая схема, отражая функции модулей, одновременно показывает структуру связей между ними. Иерархические структуры системы характеризуются, с одной стороны, вертикальным управлением, когда модули верхнего уровня имеют право вмешательства и координирования работы модулей нижнего уровня. С другой стороны, действия модулей верхнего уровня зависят от информации, полученной в результате функционирования нижних иерархических уровней. Таким образом, сверху вниз идут в основном управляющие воздействия, а снизу вверх – информация о соответствующих решениях и переменных.

Архитектура программного обеспечения формируется из ряда компонентов, соединенных различными средствами зависимости и связи. Архитектурный компонент может быть определен поразному в зависимости от архитектурного подхода и степени подробности описания архитектуры. Наиболее критическими компонентами архитектуры программного обеспечения являются компоненты, к которым происходят частые обращения, или компоненты архитектурно связанные (через зависимости и связи) со множеством других компонентов, влияя таким образом на их надежность. Процесс – компонент, который занимает слот в таблице процессов процессора. Процессом может также быть компонент, который управляется как процесс через использование примитивов операционной системы. Модуль состоит из нескольких файлов, логически связанных и образующих часть функционального кода. Размер модуля может изменяться в широком диапазоне и зависит от созданного прикладного программного обеспечения.

В зависимости от свойств архитектуры компоненты программного обеспечения, а также их связи и зависимости могут быть определены поразному. Рассматриваются следующие свойства архитектуры:

– статическая и динамическая зависимость компонентов;
– интерфейсы компонентов;
– связь и управление;
– загруженность системы.

Архитектура ПО со статической зависимостью компонентов формируется как дерево систем, подсистем, модулей и функций. При этом подходе неисправность в узле более высокого уровня может привести к недоступности узлов более низких уровней. Эта архитектура со статической зависимостью компонентов не включает зависимость между динамическими процессами, таким образом, информации относительно использования компонентов ПО в течение обработки запросов пользователей, администрирования и сопровождения модель не дает.

При динамической зависимости компонентов программная архитектура формируется как граф процессов, которые используют подсистемы, модули и функции. Процессы могут выполняться неоднократно с использованием тех же самых подсистем, модулей и функций. Процесс может использовать несколько различных функций в течение своего выполнения.

Интерфейсы компонентов включают описание взаимодействия между различными компонентами, а также интерфейсы внутри компонентов ПО, помещенных в различные аппаратные составляющие в соответствии с аппаратными конфигурациями. Интерфейсы компонентов зависят от распределения программного обеспечения и от связей, наложенных аппаратной архитектурой.

Связь и управление включают связь (в смысле сообщений), имеющуюся между процессами при обработке запросов пользователя, а также при сопровождении и администрировании. Вследствие того, что различные компоненты программного обеспечения помещены в аппаратные компоненты, которые взаимосвязаны, аппаратная архитектура также воздействует и на связь, и на управление.

Загруженность системы зависит от количества и типа запросов пользователя, а также от процедур администрирования и сопровождения. Загруженность системы отражается количеством и типом процессов и сообщений и использованием компонентов ПО. Зависимости компонентов позволяют неисправности распространяться из компонента, в котором она происходит, к другим компонентам. Это распространение может вызывать сбои в цепочке (или в дереве) компонентов. Обнаружение отказов зависит от выполненных тестов или от количества и типа запросов пользователя. Ошибка может произойти в любом компоненте. Эта ошибка может быть вызвана сбоем, переданным другим компонентом, или это может быть сбой, произошедший именно в этом компоненте. Ошибка может быть прослежена через цепочку (или дерево) зависимости компонентов для устранения всех сбоев, которые связаны с этой ошибкой.

Надежность ПО зависит от уровня, соответствующего различным компонентам и их зависимостям. В зависимости от того, где произошел сбой, длительность отказа системы и его влияние на надежность системы различны. Сбой может происходить на различных уровнях архитектуры, в модуле, процессе, интерфейсе компонента или в связи и механизме контроля.

Число архитектурных уровней в модели архитектуры ПО зависит от частного (прикладного) проекта системы. Каждый архитектурный уровень содержит граф компонентов каждого типа (граф модулей и граф процессов). Процессы используют модули в период времени их исполнения. Процесс может использовать несколько различных модулей, а модуль может использоваться несколькими различными процессами. Граф процессов представляет динамическую зависимость компонентов, а граф модулей представляет статическую зависимость компонентов. Как показано в работе [25], эта архитектура может быть расширена до произвольного числа уровней с графами компонентов на различных уровнях. В модели архитектуры граф процессов находится на самом верхнем уровне, а графы остальных компонентов, например функции, примитивы, данные, структуры и сообщения, находятся на отдельных нижних уровнях.

В крупных современных системах обработки информации часто используются СУБД как компонент архитектуры. Быстродействие и надежность СУБД значительно влияют на надежность всей системы, поэтому оптимизации быстродействия СУБД должно уделяться больше внимания на всех стадиях разработки ПО [53].

Среднее время простоя системы в архитектуре ПО зависит от условных и безусловных вероятностей сбоев на всех уровнях архитектуры и от среднего времени доступа, анализа и восстановления сбойных компонентов. Время устранения сбоя равно времени, которое требуется для доступа, анализа, восстановления. Это означает, что время восстановления меньше, чем время устранения сбоя. Если используется автоматическое восстановление и компонент не содержит сбоев, то он рассматривается как восстановленный компонент. Ошибка того же типа, что была устранена, считается новой ошибкой.

Среднее время простоя системы вычисляется для всех архитектурных уровней и всех компонентов на каждом уровне. Для каждого архитектурного уровня в ПО вероятность использования каждого компонента зависит от вероятности сбоя компонента и средних времен анализа, доступа и восстановления для этого компонента.

Более того, сбойный компонент может вызывать сбои в зависящих от него компонентах как на других уровнях архитектуры, так и на том же самом уровне. Поэтому для каждого отдельного уровня архитектуры и для всех компонентов надо учитывать условную вероятность появления сбоя и относительные времена доступа, анализа и восстановления этих компонентов. Также для одного уровня и для всех компонентов условная вероятность появления сбоя зависит от относительных времен доступа, анализа и восстановления этих компонентов.

Среднее время появления сбоя зависит от условных и безусловных вероятностей сбоев во всех

компонентах на всех архитектурных уровнях и от относительного времени использования компонентов, в которых сбоя не происходит. Среднее время сбоя вычисляется для всех архитектурных уровней и всех компонентов на каждом архитектурном уровне.

В каждом отдельном архитектурном уровне и для всех компонентов условная вероятность работы без сбоев зависит от относительного времени использования этих компонентов.

Время появления сбоя (MTTF) и среднее время простоя системы (TR) характеризуют возможность архитектуры ПО обеспечивать потенциальную производительность и достигать эту производительность после отказа. Первый показатель связан с понятием отказа ПО, а второй – с понятием восстановления. Время появления сбоя позволяет пользователю оценить возможность решения той или иной задачи. Среднее время простоя информирует о том, когда отказавшая система будет восстановлена. Время появления сбоя и среднее время простоя системы характеризуют поведение архитектуры ПО на начальном этапе работы вычислительной системы. Эти показатели не информативны при оценке работы архитектуры в течение длительного времени функционирования.

Готовность системы к эксплуатации в данный момент времени определяется как вероятность того, что система нормально функционирует в данный момент времени.

Надежность программного обеспечения определяется как вероятность того, что программное обеспечение функционирует без сбоев, в определенной операционной среде, в течение определенного промежутка времени. Коэффициент надежности архитектуры ПО можно оценить по формуле [61]

$$R = \sum_{j=1}^{M} \sum_{i=1}^{N_j} PU_{ij} R_{ij} , \qquad (1)$$

где M – число уровней архитектуры ПО; N_j – число компонент на уровне j, $j = 1, ..., M$; PU_{ij} – вероятность использования компонента; R_{ij} – надежность компонента.

Оценить данные параметры на фазе разработки архитектуры невозможно, численные показатели в модель берутся непосредственно на фазах кодирования и тестирования.

Фаза кодирования ПО. Количество ошибок в выполнимом коде напрямую зависит от ошибок, сделанных на стадии кодирования. Одним из самых часто используемых параметров оценки корректности ПО является плотность ошибок.

Начальную плотность ошибок можно оценить как [3]

$$D = C \cdot F_{ph} \cdot F_{pr} \cdot F_m \cdot F_s, \qquad (2)$$

где F_{ph} – коэффициент фазы тестирования; F_{pr} – коэффициент командного программирования; F_m – коэффициент опытности и «зрелости» процесса разработки ПО; F_s – коэффициент структурирования; C – константа, определяющая количество ошибок/KLOC (ошибок на тысячу строк исходного кода).

Коэффициенты F_{pr}, F_m, F_s и C зависят только от мастерства и опыта команды разработчиков.

При оценке коэффициента командного программирования (F_{pr}) плотность ошибок зависит от конкретных людей, их опыта написания программ и отладки. Можно принять следующие значения параметра: «Высокий», «Средний», «Низкий». Числовые показатели определяются и задаются экспертом.

Для коэффициента опытности и «зрелости» процесса разработки ПО (F_m) принимаются следующие значения параметра: «Уровень 1», «Уровень 2», «Уровень 3», «Уровень 4», «Уровень 5». Числовые показатели определяются и задаются экспертом.

Коэффициент структурирования (F_s) позволяет взять во внимание зависимость плотности ошибок от языка программирования (отношение количества кода на ассемблере и языка высокого уровня):

$$F_s = 1 + 0{,}4a, \qquad (3)$$

где a – отношение количества кода на Ассемблере и языка высокого уровня. Предполагается, что код на ассемблере может содержать на 40 % ошибок больше.

Фаза тестирования ПО. Фаза тестирования самая продолжительная и может занять до 60 % от всего проекта. Во время тестирования выявляется самое большое количество ошибок в ПО. Рассмотрим меры покрытия теста, используемые на данной фазе:

– покрытие выражений – доля всех условных выражений, выполненных во время теста;

– покрытие ветвлений – доля всех ветвлений, выполненных во время теста;

– покрытие предикатов – доля всех переменных, которые используются для проверки условий перед условным переходом.

В начальной плотности ошибок (2) коэффициент фазы тестирования (F_{ph}) принимает следующие значения [3]: «Тестирование модуля», «Подсистемы», «Системы», «Приемлемость». Числовые показатели определяются и задаются экспертом.

Параметры для оценки D по выражению (2) должны быть скорректированы с использованием данных, накопленных в результате деятельности разработчиков ПО. Коэффициент C обычно лежит в диапазоне от 6 до 20 ошибок/KLOC. Можно брать как средние значения, так и max и min значения для оценки диапазона плотности ошибок.

Для тестирования ПО берется набор выходов программы и наблюдается реакция системы. Если ответ отличен от ожидаемого, то ПО имеет хотя бы одну ошибку. Тестирование преследует две цели: увеличить надежность так быстро, как это возможно, и найти максимальное количество ошибок. С другой стороны, в течение испытаний цель состоит в том, чтобы оценить надежность, таким образом, уровень найденных ошибок должен соответствовать фактическому.

Методы тестирования программных средств можно разделить на два основных класса [14]:

1. «Черный ящик» (функциональное тестирование). Для тестирования рассматриваются только входы и выходы ПО. Внутренняя структура ПО во внимание не берется. Это наиболее общий способ тестирования.

2. «Белый ящик» (структурное тестирование). Для теста используются знания о структуре ПО.

На практике комбинация двух методов приводит к наилучшим результатам. Тестирование ПО как «черный ящик» требует функционального описания программы, хотя некоторая информация о структуре позволяет тестировщикам выбрать наилучшие параметры для входов системы. Выбор входов может происходить случайным образом или разбиваться на группы. Группы входов могут подвергаться более тщательному тестированию. Комбинирование двух способов выбора входных диапазонов тоже применяется для тестирования ПО.

Некоторые ошибки достаточно легко обнаружить. Это ошибки, имеющие высокий уровень *обнаружимости*. Ошибки, которые трудно обнаружить, называются ошибками с низким уровнем обнаружимости. Они появляются при очень редко встречающихся комбинациях значений входных диапазонов. В начале тестирования большое количество ошибок имеет высокий уровень обнаружимости. Они легко обнаруживаются и устраняются. На последующих стадиях остаются ошибки, имеющие низкий уровень обнаружимости.

Тщательность тестирования может быть измерена с помощью коэффициента полноты покрытия теста. Коэффициент покрытия ветвлений кода – более четкая мера, чем коэффициент покрытия выражений. Некоторые разработчики используют коэффициент покрытия ветвлений, равный 0,85, как минимальное значение коэффициента.

Чтобы оценить операционную (транзакционную) надежность ПО, тестирование должно быть выполнено в соответствии с операционными профилями. Операционный профиль – множество несвязанных и непересекающихся входных диапазонов и вероятностей их использования компонентами. Для разных ОС профили могут различаться. Получение операционного профиля требует разделения пространства входных диапазонов на подмножества (листья) и затем оценки вероятностей использования каждого подмножества (листа). Подмножество с достаточно высокой вероятностью использования может быть разделено на зоны меньшего размера.

Моделирование роста надежности программного обеспечения. Средства для обеспечения требуемого уровня надежности ПО могут достигать 60 % ресурсов проекта. Следовательно, тестирование должно быть тщательным образом спланировано для реализации проекта к заданной дате. Даже после длительного периода тестирования дополнительное тестирование может выявить новые ошибки. ПО как результат проекта обладает должным уровнем надежности, но содержит ошибки. Для планирования и принятия решений используются SGRM (Software Growth Reliability Model) – модели роста надежности ПО [14]. В модели SGRM предполагается, что надежность растет пропорционально времени тестирования, которое может быть измерено временем использования процессора. Рост надежности определяют в терминах интенсивности сбоев $\lambda(t)$ в зависимости от количества ожидаемых ошибок за время t $\mu(t)$, как [3]:

$$\lambda(t) = \frac{d}{dt}\mu(t) \tag{4}$$

Пусть количество ошибок за время t – $N(t)$. Предположим, что ошибки устраняются по мере обнаружения. За основу возьмем экспоненциальную модель. Предполагается, что количество найденных и исправленных ошибок пропорционально количеству существующих ошибок.

Можно показать, что β_1 определяется как

$$\beta_1 = \frac{k}{SQ\frac{1}{r}}, \tag{5}$$

где S – количество инструкций в коде; Q – количество объектных инструкций в каждой инструкции кода; r – уровень выполнения инструкции компьютером k – называется коэффициентом подверженности ошибкам и меняется от 1×10^{-7} до 10×10^{-7}; t измеряется в секундах выполнения процессорного времени:

$$N(t) = N(0)\exp(-\beta_1 t), \tag{6}$$

где $N(0)$ – начальное количество ошибок; общее количество ошибок за время t:

$$\mu(t) = N(0) - N(t) = N(0)(1 - \exp(-\beta_1 t)), \tag{7}$$

в общем случае получаем

$$\mu(t) = \beta_0(1 - \exp(-\beta_1 t)), \tag{8}$$

где β_0 – общее количество ошибок, которые могут быть обнаружены, равно $N(0)$. Это предполагает, что во время отладки не делаются новые ошибки.

Выражение для интенсивности сбоев использует равенство

$$\lambda(t) = \beta_0\beta_1\exp(-\beta_1 t). \tag{9}$$

Экспоненциальная модель легка для понимания и применения. Преимущество этой модели в том, что параметры β_0 и β_1 четко определены еще до начала тестирования.

Подход SGRM может применяться при двух различных ситуациях.

1. До начала теста. Часто надо иметь предварительный план тестирования. Для экспоненциальной или логарифмической модели можно оценить время для достижения требуемого значения интенсивности сбоев, MTTF или плотности ошибок.

Во многих случаях t должно измеряться в человеко-часах и должно быть умножено на

соответствующий коэффициент, который определяется с использованием опыта предыдущих проектов.

2. Во время теста. Используя SGRM, можно оценить дополнительное время тестирования, необходимое для достижения желаемого уровня надежности.

Основные шаги использования SGRM:

1. Сбор данных. Данные об интенсивности сбоев включают множество погрешностей и «шума». Данные часто нуждаются в сглаживании. Наиболее общая форма сглаживания – это использование группировки данных. Группировка включает разделение теста на интервалы, затем рассчитывается средняя величина интенсивности сбоев на каждом интервале.

2. Выбор модели и определение параметров. Лучший способ выбора модели – это положиться на прошлый опыт проектов, использующих такую методику. Экспоненциальная и логарифмическая модели для экстраполяции параметров используются чаще всего. Данные первых тестов содержат большое количество шума. На самых ранних стадиях тестирования параметры могут очень сильно отклоняться от реальных, и они не могут использоваться, пока не будут стабилизированы.

3. Выполнение анализа о количестве дополнительных тестов. Используя подходящую модель, мы можем оценить количество необходимых дополнительных тестов для достижения определенного уровня интенсивности сбоев. Из этого рассчитывается минимальное время, которое необходимо затратить на тестирование.

В модели SGRM предполагается, что во время тестирования используется единая методика поиска ошибок. Каждая новая стратегия изначально эффективна для поиска определенных классов ошибок, но дает «всплески» в интенсивности сбоев. Для этого надо использовать процедуру сглаживания. Большей проблемой является то, что ПО продолжает модифицироваться во время тестов. Если изменения были значительны, то приходится выбрасывать из выборки значения, полученные ранее.

Одним из важных параметров эффективности SGRM является коэффициент покрытия ошибок. Коэффициент покрытия ошибок C_d линейно зависит от $C_в$ – коэффициента покрытия ветвлений. Коэффициент покрытия ветвления показывает, насколько эффективно были «захвачены» все возможные пути ветвления программы:

$$C_d = -a + bC_в,\ C_в > 0. \qquad (10)$$

Значения параметров a и b зависят от размера ПО и начальной плотности ошибок. Преимущество использования коэффициентов покрытия в том, что они напрямую зависят от того, насколько тщательно программа исследуется. Для достижения высокой надежности ПО должна использоваться более строгая мера, такая как коэффициент покрытия предикатов. Предикаты – условные операторы, которые влияют на внутреннее поведение программы.

Операционные профили тестирования. Вполне очевидно, что абсолютно полный тест некоторых компонентов ПО практически невозможно выполнить [35]. Это легко представить на примере модуля, находящего корни квадратного уравнения. Входом в модуль является вектор коэффициентов уравнения, а выходом – вектор решений. Каждый входящий элемент может принимать значения от $-\infty$ до $+\infty$. Сразу сталкиваемся с тем ограничением, что компьютерные программы не умеют работать с бесконечными числами, т. е. при очень больших или очень малых значениях входных коэффициентов мы даже не можем вычислить решение. Другое ограничение на время тестирования – если, например, перебирать значения коэффициентов от –5000 до 5000 с шагом 0,0001, то получим, что необходимо выполнить 10^{30} запусков модуля, что может занять несколько лет при условии, что не будет обнаружено ошибок.

Некоторыми исследователями предлагается составлять так называемые *операционные профили*. Профили представляют собой таблицы, включающие информацию о диапазонах входных данных модуля и вероятности сбоя на данном диапазоне данных. Субъективные и методологические оценки процесса разработки ПО не должны отображаться в операционных таблицах данных. Конечно, класс и технологии разработчика должны учитываться при выборе компонента, но их роль должна подкрепляться точными техническими данными.

Есть два способа оценки качества ПО:

1. Компонент был признан корректным.

2. Компонент был действительно (случайным образом) протестирован.

Оба способа требуют формальную спецификацию того, что компонент должен делать. Утверждение о том, что компонент был признан корректным, это гарантия его выполнения согласно его спецификации. Заявление, говорящее, что надежность компонента выше $1 - 10^{-4}$ за одно выполнение с доверительной вероятностью 99 %, означает, что не более чем в одном из ста случаев ошибка может произойти в одном из 10 000 испытаний.

Отображения профиля. Рабочие операционные профили должны рассматриваться при занесении результатов измерений в таблицу. Так как разработчик компонента не может знать, как его продукт будет использован и, следовательно, с каким профилем он встретится, будучи частью системы, таблица данных должна рассматривать профиль как параметр. Это значит, что в ней определяется отображение профиля в параметры надежности.

Подобласти компонентов. Компонент имеет естественное разделение его входного пространства на функциональные подобласти, и практическое описание его профиля как вектора весов над этими подобластями. Такая форма профиля позволяет разработчику тестировать компонент, не зная заранее, с какими данными он может позднее встретиться.

Процесс разработки компонента и системы ПО. В общих чертах процесс создания и использования компонентов ПО можно представить так:

1. Разработчик компонента определяет множество входных подобластей для выполнения компонента.

2. Разработчик компонента измеряет свойства компонента для каждой подобласти.

3. Разработчик компонента размещает в таблице данных список подобластей компонента и надежность.

4. Системный проектировщик определяет структуру системы, использующей компоненты.

5. Используя таблицы предполагаемых компонентов и испытательных профилей для системы, системный проектировщик вычисляет надежность системы.

6. Если нельзя достигнуть требуемой надежности системы, необходимо найти лучшие компоненты или изменить структуру системы, а после этого повторить вычисления.

Такая идеализированная модель не применима на практике. В настоящее время не существует прикладной теории надежности программной системы, состоящей из компонентов. Успешное теоретическое обоснование должно использовать сравнительно простую модель, но в то же время пользоваться сложившейся теорией тестирования и теорией надежности.

Рабочие профили компонент программного обеспечения отображают информацию о качестве компонента в его таблице данных статистического характера, следовательно, она должна быть получена путем случайного тестирования. Профиль, предназначенный для компонента, зависит не только от входного профиля системы, но также от позиции компонента в системе и действий других компонентов. Операционный профиль можно записывать с помощью двух таблиц, содержащих отображения профиля (одна – для параметра надежности, другая – для изменения профиля). Эти таблицы определяются через разделение на части входной области. Системный проектировщик может использовать эти части для вычисления надежности системы до начала ее работы.

Размер компонентов может качественно повлиять на концепцию разработки ПО, основанного на компонентах. Размер компонента во внимание не берется, кроме неявного допущения о том, что разработчик компонента протестировал компонент для подготовки его операционного профиля. ПО, соответствующее по размерам операционной системе (или даже компилятора), практически не может быть протестировано в целом, так же как не могут быть определены подходящие подобласти. Из этого следует, что такие системы должны быть построены из компонентов меньшего размера, подходящего для прямых оценок. Прежде чем рассматривать операционную систему в качестве компонента, нужно проанализировать ее собственные компоненты.

Оценка и анализ надежности должны осуществляться на каждом шаге создания отказоустойчивого ПО. В связи с тем, что нет единого подхода, методики и параметров для оценки надежности ПО, в настоящее время существует множество моделей и алгоритмов, проверенных практически, но имеющих ряд недостатков. Основной недостаток всех моделей – «узкая специализация». Модели всегда привязаны к какой-либо одной фазе разработки ПО, а иногда имеют чисто теоретическую ценность из-за того, что в них должны быть использованы данные, полученные на других стадиях разработки ПО, а авторы таких моделей, как правило, отводят этому факту второстепенное значение.

Наиболее перспективным является подход, направленный на создание некоторой гибридной модели, включающей в себя все достоинства хорошо проверенных моделей и алгоритмов. Существующие модели и алгоритмы необходимо модифицировать таким образом, чтобы выходные данные одной модели стали входными для других; предусмотреть единство понимания параметров, пересчет единиц измерения, нормализацию данных и удовлетворение современным требованиям и тенденциям, например таким, как объектно-ориентированный анализ и программирование. Из данного обзора мы видим, что очень важно использовать как можно больше априорной информации перед началом каждой фазы создания ПО, так как это обеспечит наиболее приемлемые и точные результаты.

Библиографические ссылки

1. Липаев В. В. Обеспечение качества программных средств. Методы и стандарты. М. : СИНТЕГ, 2001. – 380 с.

2. Липаев В. В. Надежность программных средств. М. : СИНТЕГ, 1998.

3. Lyu M. R. Software Fault Tolerance. Published by John Wiley & Sons Ltd, 1996.

4. Ковалев И. В., Золотарёв В. В., Жуков В. Г., Жукова М. Н. Методика построения модели безопасности автоматизированных систем // Программные продукты и системы. 2012. № 2. С. 16.

5. Авиженис А. Н., Лапри Ж.-К. Гарантоспособные вычисления: от идей до реализации в проектах // ТИИЭР. 1986. Т. 74, № 5. С. 8–21.

6. Avizienis A. The N-Version approach to fault-tolerant software // IEEE Trans. on Software Engineering. 1985. Vol. SE11, № 12. P. 1491–1501.

7. Tai A., Meyer J., Avizienis A. Performability Enhancement of Fault-Tolerant Software // IEEE Trans. on Reliability. 1993. Vol. 42, No. 2. P. 227–237.

8. Липаев В. В. Тестирование компонентов и комплексов программ. М. : СИНТЕГ, 2010. 400 с.

9. Характеристики качества программного обеспечения / Б. Боэм [и др.] М. : Мир, 1981. 208 с.

10. Боэм Б. У. Инженерное проектирование программного обеспечения: пер. с англ. М. : Радио и связь, 1985. 512 с.

11. Boehm B. W., Haile A. C. Information Processing // Data Automation Implications of Air Force Command and Control Requirements in the 1980's (CCPI – 1985). Vol. I. Highlights, Report SAMSO/XRS-71-1. U. S. Air Force Systems Command (NTIS: AD 900031L). Los Angeles, CA, 1982.

12. Ashrafi N., Berman O. Optimization Models for Selection of Programs, Considering Cost & Reliability // IEEE Transaction on reliability. 1992. Vol. 41, No 2. P. 281–287.

13. Zahedi F., Ashrafi N. Software reliability allocation based on structure, utility, price, and cost // IEEE Trans. on Software Engineering. 1991. Vol. 17, No. 4. P. 345–356.

14. Lyu M., Chen J. H., Avizienis A. Software diversity metrics and measurements // In Proc. IEEE COMPSAC 1992. Chicago, 1992. C. 69–78.

15. Elmendorf W. Fault-tolerant programming // In Digest of 2-nd FTCS. Newton, MA, 1972. C. 79–83.

16. Липаев В. В. Экономика производства сложных программных продуктов. М. : СИНТЕГ, 2008. 432 с.

17. Липаев В. В., Серебровский Л. А. Технология проектирования комплексов программ АСУ. М. : Радио и связь, 1983. 264 с.

18. Липаев В. В. Тестирование программ. М. : Радио и связь, 1986. 234 с.

19. Ковалев И. В. Система мультиверсионного формирования программного обеспечения управления космическими аппаратами : дис. … д-ра техн. наук. Красноярск : КГТУ, 1997. 228 с.

20. Ковалев И. В., Юнусов Р. В. Мультиверсионный метод повышения программной надежности информационно-телекоммуникационных технологий в корпоративных структурах // Телекоммуникации и информатизация образования. 2003. № 2. С. 50–55.

21. Kovalev I. V., Dgioeva N. N., Slobodin M. Ju. The mathematical system model for the problem of multiversion software design // Proceedings of Modelling and Simulation, MS'2004 AMSE : Intern. Conf. on Modelling and Simulation, MS'2004. AMSE, French Research Council, CNRS, Rhone-Alpes Region, Hospitals of Lyon. Lyon-Villeurbanne, 2004.

22. Ковалев И. В., Царев Р. Ю., Капулин Д. В. Архитектурная надежность программного обеспечения информационно-управляющих систем / М-во сельского хоз-ва Российской Федерации; Красноярский гос. аграрный ун-т. Красноярск, 2011.

23. Ковалев И. В., Новой А. В., Штенцель А. В. Оценка надежности мультиверсионной программной архитектуры систем управления и обработки информации // Вестник СибГАУ. 2008. № 3. С. 50–52.

24. Ковалев И. В., Новой А. В. Расчет надежности отказоустойчивых архитектур программного обеспечения // Вестник СибГАУ. 2007. № 4. С. 14–17.

25. Ковалев И. В., Царев Р. Ю., Завьялова О. И. Анализ архитектурной надежности программного обеспечения информационно-управляющих систем // Приборы. 2010. № 11. С. 24–26.

26. Антамошкин А. Н., Ковалев И. В., Царев Р. Ю. Математическое и программное обеспечение отказоустойчивых систем управления и обработки информации / М-во сельского хоз-ва Российской Федерации; Красноярский гос. аграрный ун-т. Красноярск, 2011.

27. Choi, J. G., Kang H. G. Reliability Estimation of nuclear digital I&C systems using Software Functional Block Diagram and control flow. FastAbstract ISSRE Copyright, 2000.

28. Berman O., Cutler M. Choosing an Optimal Set of Libraries // IEEE Transaction on reliability. 1996, Vol. 45, No 2. P. 303–307.

29. Choi J. G., Kang H. G. Reliability Estimation of nuclear digital I&C systems using Software Functional Block Diagram and control flow. FastAbstract ISSRE Copyright, 2002.

30. Costa D., Mendez T. On the Extention of Exception to Support Software Faults Models. FastAbstract ISSRE Copyright, 2000.

31. David Ph., Guidal C. Development of a fault tolerant computer system for the Hermes Space Shuttle // IEEE Trans. 1993. P. 641–648.

32. Dunham J. R., Knight C. J. Production of reliable flight crucial software: Validation method research for fault-tolerant avionics and control systems sub-working-group meeting // NASA Conf. Pub. 2222. NASA, 1985.

33. Grams T. The Poverty of Reliabiliy Growth Models. FastAbstract ISSRE Copyright, 1999.

34. Goseva-Popova K., Trivedi K. S., Mathur A. P. How Different Architecture Based Software Reliability Models are Reealated. FastAbstract ISSRE Copyright, 2000.

35. Hamlet D., Mason D., Wiot D. Foundational Theory of Software Component Reliability. FastAbstract ISSRE Copyright, 2000.

36. Hecht H. Fault tolerant software // IEEE Trans. Reliability. Vol. R-28. 1979. P. 227–232.

37. Hui-Qun Z. A., Jing S., Yuan G. New Method for Estimating the Reliability of Software System Based on Components. FastAbstract ISSRE and Chillarege Corp. Copyright, 2001.

38. Hudak J., Suh B.-H., Sieweorek D., Segall Z. Evaluation & comparition of fault-tolerant software techniques // IEEE Trans. Reliability. 2009. Vol. 14. P. 1229–1237.

39. Karunanithi N., Whitley D., Malaiya Y. K. Prediction of Software Reliability Using Connectionist // IEEE transactions on reliability. Models July. 1992. Vol. 18, No. 7.

40. Kaszycki G. Using Process Metrics to Enhance Software Fault Prediction Models. FastAbstract ISSRE Copyright, 1999.

41. Keene S. Progressive Software Reliability Modeling. FastAbstract ISSRE Copyright, 1999.

42. Knight C. J., Levenson N. G. An experimental evaluation of the assumption of independence in Multiversion programming // IEEE Trans. Software Engineering. 1986. Vol. SE-12. P. 96–109.

43. Levendel Y. Reliability analysis of large software systems: Defect data modeling // IEEE Trans. Software Engineering. 1990. Vol. 16. P. 141–152.

44. Liestman A., Campbell R.-H. Fault-Tolerant Scheduling Problem // IEEE Trans. on Software Engineering. 1986. Vol. SE-12. P. 1089–1095.

45. Pai G. J., Dugan J. B. Enhancing Software Reliability Estimation Using Bayaesan Network and Fault Trees. FastAbstract ISSRE and Chillarege Corp. Copyright, 2001.

46. Shooman M. L. Software Reliability for Use During Proposal and Early Design Stages. FastAbstract ISSRE Copyright, 2009.

47. Wattanapongsakorn N. Reliability Optimization for Software Systems with Multiple Applications. FastAbstract ISSRE and Chillarege Corp. Copyright, 2001.

48. Xie M., Yang B. Regression Goodness-Of-fit Test for Software Reliability Model Validation. FastAbstract ISSRE Copyright, 2000.

49. Lyu M. R. Handbook of Software Reliability Engineering / Edited by Michael R. Lyu Published. IEEE Computer Society Press and McGraw-Hill Book Company, 1996. 819 p.

50. Rosenberg L., Hammer T., Shaw J. Software Metrics and Reliability // Best Paper : Software reliability engineering was presented at the 9-th International Symposium, Award, 1998.

51. Ковалев И. В., Слободин М. Ю., Ступина А. А. Математическая постановка задачи проектирования *n*-версионных программных систем // Проблемы машиностроения и автоматизации. 2005. № 3. С. 16–23.

52. Ковалев И. В., Царев Р. Ю., Русаков М. А., Слободин М. Ю. Модели поддержки многоэтапного анализа надежности программного обеспечения автоматизированных систем управления // Проблемы машиностроения и автоматизации. 2005. № 2. С. 30–35.

53. Ковалев И. В., Ступина А. А., Гаврилов Е. С. Транзакционная надежность технологий обработки данных в информационно-управляющих системах // Вестник СибГАУ. 2005. № 3. С. 52–57.

54. Engel E. A., Kovalev I. V. Information processing using intelligent algorithms by solving wcci 2010 tasks // Вестник СибГАУ. 2011. № 3. С. 4–8.

55. Головкин Б. А. Расчет характеристик и планирование параллельных вычислительных процессов. М. : Радио и связь, 1983. 272 с.

56. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов : пер. с англ. М. : Мир, 1981. 366 с.

57. Мамиконов А. Г., Кульба В. В., Косяченко С. А. Типизация разработки модульных систем обработки данных. М. : Наука, 1989. 165 с.

58. Мамиконов А. Г., Кульба В. В. Синтез оптимальных модульных систем обработки данных. М. : Наука, 1986.

59. Михалевич В. С., Волкович В. Л. Вычислительные методы исследования и проектирования сложных систем. М. : Наука, 1982. 286 с.

60. Ковалев И. В., Нургалеева Ю. А., Ежеманская С. Н., Ерыгин В. Ю. Многоатрибутивное управление трудозатратами на разработку *n*-вариантных программных систем // Фундаментальные исследования. 2011. № 8–1. С. 124–127.

61. Kovalev I. V., Younoussov R. V. Fault-tolerant software architecture creation model based on reliability evaluation // Advanced in Modeling & Analysis : Journal of AMSE Periodicals. 2002. Vol. 48, № 3–4. P. 31–43.

References

1. Lipaev V. V. *Obespechenie kachestva programmnyh sredstv* [Quality assurance software]. Moscow, Nauka Publ., 2001, 380 p.

2. Lipaev V. V. *Nadezhnost' programmnyh sredstv*. [The reliability of software]. Moscow, Nauka Publ., 1998.

3. Lyu M. R. Software Fault Tolerance. Edited by Michael R. Lyu Published by John Wiley & Sons Ltd, 1996.

4. Kovalev I. V., Zolotarjov V. V., Zhukov V. G., Zhukova M. N. [A method of constructing models of security of automated systems]. *Programmnye produkty i sistemy*. 2012, no. 2, p. 16. (In Russ.)

5. Avizhenis A. N., Lapri. Zh.-K. [Dependable computing: from ideas to implementation projects]. *TIIER*, 1986., vol. 74, no. 5, p. 8–21. (In Russ.)

6. Avizienis A. The N-Version approach to fault-tolerant software. *IEEE Trans. on Software Engineering.* 1985, Vol. 11, no. 12, p. 1491–1501.

7. Tai A. Performability Enhancement of Fault-Tolerant Software. *IEEE Trans. on Reliability.* 1993, Vol. 42, no. 2, p. 227–237.

8. Lipaev V.V. *Testirovanie komponentov i kompleksov programm* [Testing of components and systems programs]. Moscow, SINTEG Publ., 2010, 400 p.

9. Bojem B. *Harakteristiki kachestva programmnogo obespechenija* [Characteristics of software quality]. Moscow, Mir Publ., 1981, 208 p.

10. Bojem B. U. *Inzhenernoe proektirovanie programmnogo obespechenija* [Engineering design software]. Moscow, Radio i svjaz Publ., 1985, 512 p.

11. Boehm B. W. Information Processing. Data Automation Implications of Air Force Command and Control Requirements in the 1980's (CCPI – 1985), Vol. I: Highlights, Report SAMSO/XRS-71-1, U.S. Air Force Systems Command (NTIS: AD 900031L), Los Angeles, CA, 1982.

12. Ashrafi N. Optimization Models for Selection of Programs. Considering Cost & Reliability. *IEEE Transaction on reliability.* 1992, Vol. 41, no 2, p. 281–287.

13. Zahedi F. Software reliability allocation based on structure, utility, price, and cost. *IEEE Trans. on Software Engineering*, April 1991,Vol. 17, no. 4, p. 345–356.

14. Lyu M. Software diversity metrics and measurements. IEEE COMPSAC 1992. 1992 Chicago, p. 69–78.

15. Elmendorf W. Fault-tolerant programming. In Digest of 2-nd FTCS. June 1972, p. 79–83.

16. Lipaev V. V. *Jekonomika proizvodstva slozhnyh programmnyh produktov* [The economy of the production

of complex software products]. Moscow, SINTEG Publ., 2008, 432 p.

17. Lipaev V. V. *Tehnologija proektirovanija kompleksov programm ASU* [Design technology software systems ACS]. Moscow, Radio i svjaz Publ., 1983, 264 p.

18. Lipaev V. V. *Testirovanie programm* [Testing programs] Moscow, Radio i svjaz Publ., 1986, 234 p.

19. Kovalev I. V. *Sistema mul'tiversionnogo formirovanija programmnogo obespechenija upravlenija kosmicheskimi apparatami. Dis. dok. tehn. nauk.* [System multiversioned views of the formation of the software control of spacecraft. Dr. techn. sci.diss]. Krasnojarsk, KGTU, 1997. 228 p.

20. Kovalev I. V. [Multiversioned views method for increasing software reliability information and telecommunication technologies in corporate structures]. *Telekommunikacii i informatizacija obrazovanija.* 2003, no. 2, p. 50–55. (In Russ.)

21. Kovalev I. V., Dgioeva N. N., Slobodin M. Ju. The mathematical system model for the problem of multiversion software design. Mezhdunarodnaja konferencija po modelirovaniju i imitacii MS'2004 *AMSE.* International Conference on Modelling and Simulation, MS'2004. AMSE, French Research Council, CNRS, Rhone-Alpes Region, Hospitals of Lyon. Lyon-Villeurbanne, 2004.

22. Kovalev I. V., Carev R. Ju., Kapulin D. V. *Arhitekturnaja nadezhnost' programmnogo obespechenija informacionno-upravljajushhih system* [Architectural software reliability information management systems]. Ministry of agriculture of the Russian Federation, Krasnoyarsk state agrarian University, Krasnoyarsk, 2011.

23. Kovalev I. V., Novoj A. V., Shtencel A. V. [Reliability estimation multiversioned views of software architecture, control systems and information processing.]. *Vestnik SibGAU*. 2008, no. 3 (20), p. 50–52. (In Russ.)

24. Kovalev I. V., Novoj A. V. [Calculation of reliability of fault-tolerant software architectures.]. *Vestnik SibGAU*. 2007, no. 4 (17), p. 14–17. (In Russ.)

25. Kovalev I. V., Carev R. Ju., Zav'jalova O. I. [Analysis of architectural software reliability information management systems]. *Pribory.* 2010, no. 11, p. 24–26. (In Russ.)

26. Antamoshkin A. N., Kovalev I. V., Carev R. Ju. *Matematicheskoe i programmnoe obespechenie otkazoustojchivyh sistem upravlenija i obrabotki informacii* [Mathematical and software fault-tolerant control systems and information processing]. Ministry of agriculture of the Russian Federation, Krasnoyarsk state agrarian University, Krasnoyarsk, 2011.

27. Choi J. G. Reliability Estimation of nuclear digital I&C systems using Software Functional Block Diagram and control flow. FastAbstract ISSRE Copyright. 2000.

28. Berman O. Choosing an Optimal Set of Libraries. *IEEE Transaction on reliability.* June 1996, vol. 45, no 2, p. 303–307.

29. Choi J. G. Reliability Estimation of nuclear digital I&C systems using Software Functional Block Diagram and control flow. FastAbstract ISSRE Copyright. 2002.

30. Costa D. On the Extention of Exception to Support Software Faults Models. FastAbstract ISSRE Copyright. 2000.

31. David Ph. Development of a fault tolerant computer system for the Hermes Space Shuttle. IEEE Trans., 1993, p. 641–648.

32. Dunham J. R. Eds. Production of reliable flight crucial software: Validation method research for fault-tolerant avionics and control systems sub-working-group. NASA Conf. Pub. 2222, NASA, 1985.

33. Grams T. The Poverty of Reliabiliy Growth Models. FastAbstract ISSRE Copyright 1999.

34. Goseva-Popova K. How Different Architecture Based Software Reliability Models are Reealated. FastAbstract ISSRE Copyright. 2000.

35. Hamlet D. Foundational Theory of Software Component. FastAbstract ISSRE Copyright 2000.

36. Hecht H. Fault tolerant software. *IEEE Trans. Reliability.* 1979, vol. 28, p. 227–232.

37. Hui-Qun Z. A. New Method for Estimating the Reliability of Software System Based on Components. FastAbstract ISSRE and Chillarege Corp. Copyright. 2001.

38. Hudak J. Evaluation & comparition of fault-tolerant software techniques. *IEEE Trans. Reliability*. 2009, vol. 14, p. 1229–1237.

39. Karunanithi N. Prediction of Software Reliability Using Connectionist. *IEEE transactions on reliability Models.* July 1992, vol. 18, no. 7.

40. Kaszycki G. Using Process Metrics to Enhance Software Fault Prediction Models. FastAbstract ISSRE Copyright. 1999.

41. Keene S. Progressive Software Reliability Modeling. FastAbstract ISSRE Copyright, 1999.

42. Knight C. J. An experimental evaluation of the assumption of independence in Multiversion programming. *IEEE Trans. Software Engineering,* 1986, vol. 12, p. 96–109.

43. Levendel Y. Reliability analysis of large software systems: Defect data modeling. *IEEE Trans. Software Engineering,* 1990, vol. 16, p. 141–152.

44. Liestman A. Fault-Tolerant Scheduling Problem. *IEEE Trans. on Software Engineering*, 1986, vol. 12, p. 1089–1095.

45. Pai G. J. Enhancing Software Reliability Estimation Using Bayaesan Network and Fault. ISSRE and Chillarege Corp. Copyright, 2001.

46. Shooman M. L. Software Reliability for Use During Proposal and Early Design Stages. FastAbstract ISSRE Copyright, 2009.

47. Wattanapongsakorn N. Reliability Optimization for Software Systems with Multiple Applications. FastAbstract ISSRE and Chillarege Corp. Copyright, 2001.

48. Xie M. Regression Goodness-Of-fit Test for Software Reliability Model Validation. FastAbstract ISSRE Copyright, 2000.

49. Lyu M.R. Handbook of Software Reliability Engineering . IEEE Computer Society Press and McGraw-Hill Book Company, 1996, 819 p.

50. Rosenberg L. Software Metrics and Reliability. Software reliability engineering was presented at the 9-th International Symposium, “Best Paper” Award, November, 1998.

51. Kovalev I. V., Slobodin M. Ju., Stupina A. A. [Mathematical formulation of the problem of designing n-version software systems]. *Problemy mashinostroenija i avtomatizacii*. 2005, no. 3, p. 16–23. (In Russ.)

52. Kovalev I. V., Carev R. Ju., Rusakov M. A., Slobodin M. Ju. [Models support multi-stage analysis of software reliability of automated control systems]. *Problemy mashinostroenija i avtomatizacii*. 2005, no. 2, p. 30–35. (In Russ.)

53. Kovalev I. V., Stupina A. A., Gavrilov E. S. [Transactional reliability of data processing technology in information and control systems]. *Vestnik SibGAU*. 2005, no. 3, p. 52–57. (In Russ.)

54. Engel E. A., Kovalev I. V. [Information processing using intelligent algorithms by solving wcci 2010 tasks]. *Vestnik SibGAU*. 2011, no. 3 (36), p. 4–8. (In Russ.)

55. Golovkin B. A. *Raschet harakteristik i planirovanie parallel'nyh vychislitel'nyh processov* [Calculation of characteristics and planning of parallel computing processes]. Moscow, Radio I Svyaz Publ., 1983, 272 p.

56. Gudman S. *Vvedenie v razrabotku i analiz algoritmov* [Introduction to the design and analysis of algorithms]. Moscow, Mir Publ., 1981, 366 p.

57. Mamikonov A. G. *Tipizacija razrabotki modul'nyh sistem obrabotki dannyh* [Typing the development of modular data processing systems]. Moscow, Nauka Publ., 1989, 165 p.

58. Mamikonov A. G. *Sintez optimal'nyh modul'nyh sistem obrabotki dannyh* [Synthesis of optimal modular data processing systems]. Moscow, Nauka Publ., 1986.

59. Mihalevich V. S. *Vychislitel'nye metody issledovanija i proektirovanija slozhnyh sistem* [Computational methods for the study and design of complex systems]. Moscow, Nauka Publ., 1982, 286 p.

60. Kovalev I. V., Nurgaleeva Ju. A., Ezhemanskaja S. N., Erygin V. Ju. [Diversified management work on the development of n-variant software systems]. *Fundamental'nye issledovanija.* 2011, no. 8–1, p. 124–127. (In Russ.)

61. Kovalev I. V. Fault-tolerant software architecture creation model based on reliability evaluation. *Journal of AMSE.* 2002, vol. 48, no. 3, p. 31–43.