ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г.

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru E-mail: prin@novtex.ru

Журнал включен в систему Российского индекса научного цитирования.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

УДК 004.41(075.8)

В. В. Липаев, д-р техн. наук, проф, гл. науч. сотр.
Институт системного программирования РАН, г. Москва
e-mail: vlip28@mail.ru

Прогнозирование экономических характеристик производства заказных программных продуктов

Изложены основы прогнозирования экономических характеристик производства программных продуктов и особенности прогнозирования с использованием модели COCOMO II. Рассмотрено влияние специалистов, участвующих в создании продукта, особенности технологической и компьютерной среды на результаты прогнозирования экономических характеристик призводства. Представлены примеры прогнозирования экономических характеристик производства программных продуктов.

Ключевые слова: *прогнозирование, экономические характеристики, факторы производства, программные продукты, модель COCOMO II, примеры прогнозирования*

Основы прогнозирования экономических характеристик производства программных продуктов

При подготовке контракта на производство заказного программного продукта заказчику и менеджерам разработки необходимо оценивать его экономические характеристики, которые способны удовлетворить договаривающиеся стороны. Для небольших относительно простых проектов программных продуктов во многих случаях достаточно достоверными могут быть интуитивные оценки ***требуемых экономических ресурсов***, выполняемые опытными руководителями, реализовавшими несколько аналогичных проектов. Однако ***интуитивные оценки руководителями*** размеров и сложности крупных программных проектов, как правило, отличаются ***большими ошибками при планировании экономических характеристик***. В данном случае особенности состоят в том, что ***менеджеры и разработчики*** комплексов программ, как правило, не знают даже основ экономики промышленного производства сложной продукции, а ***экономисты современного производства*** не представляют сущность и свойства объектов разработки — программных продуктов, а также особенностей экономики технологических процессов их проектирования, производства и применения.

Начиная разработку крупного проекта, руководители, прежде всего, должны решить ***задачу целесообразности его создания*** и оценить, какова будет возможная эффективность применения готового продукта, оправдаются ли затраты на разработку и использование. Поэтому такие проекты должны начинаться с ***прогнозирования, анализа и экономического обоснования*** предстоящего жизненного цикла предполагаемого программного продукта. Для этого им следует использовать накопленный и обобщенный опыт оценивания ***реальных экономических характеристик*** производства подобных программных продуктов, а также учитывать цели их оценивания, потребности заказчиков и разработчиков в экономической информации для принятия решений на этапах проектирования и производства комплекса программ.

Задача экономического прогнозирования проектов в программной инженерии заключается, ***прежде всего, в разработке, освоении и использовании методов для оценивания экономических характеристик производства сложных программных продуктов.*** Накоплен и опубликован значительный объем полезных методических и статистических данных об экономических характеристиках создания программных комплексов разных классов. Наиболее полные исследования, обобщения и экономические характеристики реализованных проектов отражены в книгах [1, 2] и в отечественных работах [3, 4]. Анализ этих данных совместно

с результатами других авторов позволил заложить ***основы экономики*** производства сложных программных продуктов, методик прогнозирования необходимых ресурсов для разработки комплексов программ, их достоверного экономического планирования и формирования производственных процессов. Этим проблемам значительное внимание уделяется в программной инженерии при изложении практических методов производства сложных программных продуктов [5—7].

Цель экономического обоснования проектов комплексов программ состоит в помощи заказчикам и руководителям их производства определять:

— целесообразно ли проводить или продолжать работы над конкретным проектом программного продукта для детализации требований, функций и экономических характеристик или следует его прекратить ввиду недостаточных ресурсов специалистов, времени или возможной трудоемкости производства;

— при наличии достаточных ресурсов следует ли провести маркетинговые исследования для определения рентабельности полного выполнения проекта и производства программного продукта для поставки заказчику или на рынок;

— достаточно ли полно и корректно формализованы требования заказчика к проекту, на основе которых проводились оценки экономических характеристик, или их следует откорректировать и выполнить повторный анализ с уточненными исходными данными;

— есть ли возможность применить готовые повторно используемые компоненты, в каком относительном количестве от всего комплекса программ, и рентабельно ли их применять в конкретном проекте или весь проект целесообразно разрабатывать как полностью новый.

Величина и достоверность определения размера комплекса программ — ключевой фактор последующего экономического анализа, поэтому целесообразно применять несколько доступных методов для его оценивания с учетом достоверности характеристик прототипов проекта, а также по результатам предварительного проектирования. Конкретизация функций, структуры комплекса программ и состава компонентов проекта позволяет более достоверно определять размеры компонентов комплекса программ и, суммируя их, оценить размер всего комплекса. Прогнозирование совокупных затрат и влияния основных факторов позволяет избежать грубых ошибок, связанных с нерациональным планированием распределения ресурсов проекта при ***детальном проектировании и последующем производстве заказного комплекса программ.***

Обобщенные оценки экономических характеристик производства заказных комплексов программ обычно представляются в виде таблиц с указанием достоверности оценок следующих базовых характеристик, позволяющих формировать основные экономические требования в контрактах:

- полной трудоемкости производства программного продукта — C (человеко-месяцы с указанием учитываемых факторов);
- полной длительности производства программного продукта — T (месяцы);
- участвовавшего среднего числа специалистов — N (человек);
- средней производительности труда специалистов — P (число строк на человеко-месяц).

Суммарные ***затраты интеллектуального труда специалистов на производство программного продукта — трудоемкость***, является основным интегральным экономическим показателем каждого программного проекта. Эти затраты подлежат оценке и минимизации при условии обеспечения заданных заказчиком функциональных характеристик программного продукта и его качества. На ***совокупную трудоемкость*** при создании программного продукта влияет ряд факторов, при определении которых на практике используются различные единицы. Трудоемкость характеризуется временем производительного труда определенного числа специалистов, необходимого для создания программного продукта, его компонентов или выполнения определенных этапов работ в жизненном цикле. Такой подход привел к активному использованию следующих единиц трудоемкости: человеко-день, человеко-месяц, человеко-год.

Длительность производства программного продукта зависит от многих факторов и, прежде всего, от его сложности. Для учета затрат времени коллектива специалистов на конкретный комплекс программ особенно сложно фиксировать ***начало разработки***. Дело в том, что системный анализ зачастую входит в научно-исследовательские работы, финансируемые, планируемые и учитываемые независимо от начала производства конкретного программного продукта. В ряде случаев первичная разработка концепции комплекса программ является обобщением опыта создания и эксплуатации ранее разработанных, унаследованных программ. ***Окончанием разработки*** для прекращения учета затрат при оценке трудоемкости и длительности производства конкретного продукта обычно принимается успешное завершение испытаний и оформление акта пригодности для поставки и применения программного продукта соответствующей комиссией заказчика.

Длительность производства заказных программных продуктов определяет общие сроки производства сложных систем, а значит быстроту реализации идей и методов в соответствии с требованиями контракта. Однако принципиально нерентабельно производство даже очень сложных программных продуктов более трех—пяти лет. Вместе с тем, комплексы программ даже в несколько тысяч строк по полному технологическому циклу как продукция с испытаниями и документацией редко создаются за время, меньшее, чем полгода—год. Таким образом, практически длительность производства программных продуктов ***ограничена сверху и снизу***, и одним из основных факторов, определяющим эти границы, является размер комплексов программ. Их цели, концептуальная основа и алгоритмы ***не должны устареть*** за время проектиро-

вания и производства. Даже для довольно сложных программных продуктов, имеющих размер ***свыше*** 500 тыс. строк, вряд ли допустима длительность разработки более трех лет. ***Границу снизу*** определяют естественный технологический процесс коллективного производства и необходимость выполнения ряда скоординированных работ на последовательных этапах, которые обеспечивают получение продукта требуемого качества.

Число необходимых специалистов и особенности их квалификации неравномерно распределяются по этапам жизненного цикла комплекса программ. На начальных этапах системного проектирования и завершающих этапах испытаний программного продукта используется относительно небольшое число, но наиболее высококвалифицированных специалистов. На средних этапах разработки и тестирования программных модулей и компонентов в проекте участвует наибольшее число специалистов относительно невысокой квалификации. Однако для оценок и прогнозирования экономических характеристик широко применяется среднее число специалистов.

Для прогнозирования экономических характеристик при производстве новых заказных комплексов программ необходимы следующие исходные данные:

- обобщенные характеристики использованных ресурсов и экономические характеристики завершенных разработок — прототипов проектируемого комплекса программ, а также оценки влияния на эти характеристики различных факторов продуктов и среды производства;
- реализованные и обобщенные перечни выполненных работ и реальные графики проведенного ранее производства совокупности программных продуктов различных классов;
- цели и содержание частных работ в процессе производства сложных комплексов программ и требования к их выполнению для обеспечения необходимого качества программных продуктов в целом;
- структура и содержание документов, являвшихся результатом выполнения частных производственных работ.

Экономические характеристики производства реальных завершенных проектов программных продуктов собираются, накапливаются и обрабатываются с начала 80-х годов XX века в разных отечественных организациях и за рубежом [1—3]. Они позволили прогнозировать основные характеристики процессов производства сложных программных продуктов. На базе серьезных статистических исследований экономических характеристик жизненного цикла достаточно большой совокупности завершенных комплексов программ ***осуществлялось обобщение и создание теоретических и практических основ экономики производства сложных программных продуктов***. Внимание было сосредоточено на концептуальной основе распределения затрат труда в процессе производства программных продуктов, на ***факторах, определяющих реальные трудозатраты и другие экономические показатели***, а также на исследовании таких характеристик в реализованных проектах.

Прежде всего, необходимо было ***изучение реальных экономических характеристик*** проектирования и производства современных сложных программных продуктов, реализованных ***прототипов*** для прогнозирования организации и планирования новых проектов. Рассматривались преимущественно средние и крупные проекты, создаваемые большими коллективами специалистов. Вследствие этого значительно ***нивелировались индивидуальные особенности и квалификация отдельных специалистов*** и появилась возможность оценивать усредненные характеристики производительности труда и другие экономические характеристики производства в больших коллективах.

Наиболее подробно ***основные закономерности и влияние факторов на экономические характеристики*** процессов производства сложных программных продуктов в 80-е годы XX века исследовались ***за рубежом***. В 1981 г. на основе исследования процессов разработки ***63 проектов*** была опубликована модель прогнозирования экономических характеристик КОМОСТ [1]. В дальнейшем эта модель была развита, детализирована и опубликована как COCOMO, а в ***2000 г.*** под названием COCOMO II [2]. В этой модели на основе анализа более ***160 реальных проектов*** разработки программных продуктов различной сложности уточнены рейтинги влияния выделенных факторов на основные экономические характеристики производства. Обобщенные данные этих работ ниже используются и рекомендуются как базовые, для прогнозирования затрат при создании сложных заказных программных продуктов. Вместе с тем, следует отметить, что опубликованы результаты анализа экономических характеристик производства около 250 отечественных крупных программных продуктов, которые подробно изложены в монографии [3].

Однако реальные экономические характеристики производства программных продуктов публикуются очень скупо и оперативно не обобщаются, по-видимому, они считаются коммерческой тайной предприятий, которым не выгодно раскрывать фактические недостатки и неэффективность своих производств. Так как предприятия и разработчики сложных комплексов программ ***не заинтересованы раскрывать реальную экономику*** выполненных проектов и производства программных продуктов, опубликованные экономические характеристики программных проектов носят отрывочный, не упорядоченный и не всегда достоверный характер. Опубликованы ***ориентиры***, в которых средняя стоимость разработки одной строки текста программ реального времени составляет около 100 долл. США, а стоимость программ особенно высокого качества для бортовых комплексов программ системы Шатл достигает 1000 долл. США. Для административных систем ориентиром может служить стоимость строки текста программ около 20—50 долл. США.

Прогнозирование экономических характеристик производства программных продуктов с использованием модели COCOMO II

Для ***экономического прогнозирования производства конкретного программного продукта*** ниже рекомендуется и применяется наиболее известная модель COCOMO II [2, 6, 7]. Она позволяет определять основные прогнозируемые экономические характеристики, а также уточнять некоторые дополнительные данные. Для ***прогнозирования трудоемкости С проектирования и производства программных продуктов*** (человеко-месяцы) ***в модели COCOMO II*** предложены следующие выражения:

$$C = AP^E \prod_{i=1}^{n} M(i). \qquad (1)$$

Здесь P^E— размер продукта, причем $E = B + 0{,}01 \sum_{j-1}^{5} F(j)$; $M(i)$ — множители трудоемкости; $F(j)$ — факторы масштабирования; $A = 2{,}94$; $B = 0{,}91$.

В ***детальной модели COCOMO II*** на трудоемкость производства программного продукта учитывается влияние 22 факторов, представленных в таблице. Пять из них — ***масштабные факторы*** — характеризуются суммой $F(j)$ в величинах степени влияния размера комплекса программ E. В уравнении оценки трудозатрат в модели COCOMO II варьируются и применяются факторы масштабирования $F(j)$ в целях обобщения воздействия основных параметров при прогнозировании производства программных продуктов. Произведение множителей $M(i)$ (17 факторов) непосредственно влияет на изменение трудоемкости производства в выражении (1).

При использовании выражения (1) следует выбирать набор факторов (***калибровать модель***), имеющих значения коэффициентов изменения трудоемкости (КИТ) $F(j)$ в соответствии с характеристиками ***конкретного проекта***. Для выбора значений $M(i)$ в соответствии с ***характеристиками конкретного продукта и среды разработки*** следует использовать таблицы рейтингов трудоемкости для выделенных четырех групп, представленных в таблице. Номинальными (средними) в выражении (1) принимаются значения $M(i) = 1$, при которых соответствующий фактор не влияет на трудоемкость производства программного продукта. Реально для различных конкретных проектов с учетом их особенностей при калибровании используется только около половины факторов, представленных в таблице.

Для прогнозирования длительности Т производства программных продуктов в модели COCOMO II более детально рассмотрены исходные данные на основе анализа трудоемкости его производства. При этом рекомендуется учитывать такие же наборы факторов, как в выражении (1), которые применяются при прогнозировании трудоемкости непосредственного производства программного продукта C:

Состав факторов детальной модели COCOMO II

Обозначение фактора	Содержание фактора
	Масштабные факторы
$F1$	Новизна проекта
$F2$	Согласованность с требованиями и интерфейсами
$F3$	Управление рисками и архитектурой проекта
$F4$	Слаженность работы коллектива
$F5$	Технологическая зрелость обеспечения разработки
Определяется сумма рейтингов $F(j)$ для расчетов трудоемкости и длительности	
	Факторы, влияющие на затраты производства
	Требования и характеристики объекта производства
$M1$	Надежность функционирования
$M2$	Размер базы данных
$M3$	Сложность функций и структуры
$M4$	Требование повторного использования компонентов
$M5$	Полнота и соответствие документации проекта
	Характеристики коллектива специалистов
$M9$	Квалификация аналитиков
$M10$	Квалификация программистов
$M11$	Стабильность коллектива
$M12$	Опыт работы по тематике проекта
$M13$	Опыт работы в инструментальной среде
$M14$	Опыт работы с языками программирования
	Технологическая среда производства
$M15$	Уровень инструментальной поддержки проекта
$M16$	Необходимость распределенной разработки проекта
$M17$	Ограничения длительности производства продукта
	Аппаратурно-вычислительная среда производства
$M6$	Ограниченность времени исполнения программ
$M7$	Ограниченность доступной оперативной памяти
$M8$	Изменчивость виртуальной среды разработки проекта
Определяется произведение рейтингов $M(i)$ для расчетов трудоемкости и длительности	

$$T = G \times C^H, \qquad (2)$$

где $G = 3{,}67$; $H = D + 0{,}02 \times 0{,}01 \times \sum_{j-1}^{5} F(j) = D + 0{,}02 \times (E - B)$; $D = 0{,}28$.

Максимальные величины каждого из КИТ производства программных продуктов экспериментально оценены в работе [2] в предположении, что остальные параметры зафиксированы. В действительности многие факторы взаимно коррелированны. Так, например, высокой сложности комплекса программ обычно сопутствует требование высокой безопасности и надежности функционирования, а также длительная

эксплуатация. Ряд факторов отражается одновременно на нескольких КИТ.

Большую роль в повышении экономической эффективности производства программных продуктов играет ***повторное использование готовых программных компонентов*** из других проектов. При этом необходимо учитывать дополнительные затраты для обеспечения возможности повторного использования компонентов вместо их разработки каждый раз заново, а также сокращение интегральных затрат на производство продукта при их применении.

При калибровке модели COCOMO II предлагаются следующие последовательные процедуры для конкретного проекта:

- устанавливаются значения и ***сумма рейтингов*** масштабных факторов F, при которых проводится первичное оценивание экономических характеристик;
- выбирается набор, значения и оценивается ***произведение рейтингов*** факторов M, оказывающих наибольшее влияние на прогнозируемую трудоемкость производства программного продукта, которые сравниваются с предыдущей оценкой;
- для каждого набора рейтингов выбранных факторов проводится расчет и анализ трудоемкости и длительности для конкретного проекта комплекса программ.

Преимущества модели COCOMO II [2] состоят в следующем:

- возможен учет достаточно полной номенклатуры факторов, влияющих на экономические характеристики производства сложных программных продуктов;
- предложены наборы характеристик для выбора возможных рейтингов для всей номенклатуры используемых факторов;
- фактические данные подбираются в соответствии с реальными проектами комплексов программ и факторами корректировки, которые соответствуют конкретному проекту и предприятию;
- метод является достаточно универсальным и может поддерживать различные размеры, режимы и уровни качества продуктов;
- возможна высокая степень достоверности калибровки с опорой на предыдущий опыт коллектива специалистов;
- результаты прогнозирования сопровождаются обязательной документацией;
- модель относительно проста в освоении и применении.

Недостатки модели COCOMO II

- Все результаты применения модели зависят от размера программного продукта. Точность оценки такого размера оказывает ***определяющее влияние*** на точность прогноза трудозатрат.
- Результаты зависят от длительности разработки, численности специалистов и производительности труда разработчиков.
- Недостаточно учитывается внешняя среда производства и применения программного продукта.

Кроме того, модель COCOMO II представляет ***трудозатраты по сумме всех этапов производства*** — от планирования концепции до поставки программного продукта. При использовании модели COCOMO II предполагается наличие базового уровня трудозатрат, используемого при управлении конфигурацией и обеспечении должного качества. При прогнозировании экономических характеристик производства программных продуктов ***для систем реального времени*** необходимо учитывать затраты на создание моделей внешней среды, которые по величине могут быть сопоставимыми с затратами на основной программный продукт.

Наиболее значительные интегральные факторы модели COCOMO II, ***пять масштабных факторов прогнозирования*** $F(j)$, могут существенно изменять трудоемкость и стоимость производства. Остальные 17 факторов M, объединенные в ***четыре группы*** модели COCOMO II (см. таблицу), могут влиять на экономические характеристики производства сложных программных продуктов на уровне 5—10 % трудоемкости программных продуктов [2].

Новизну проекта $F1$ рекомендуется учитывать при прогнозировании по степени формализации целей и функций программного продукта, наличия у специалистов опыта аналогичных разработок, а также необходимости создания новой архитектуры, новых компонентов, алгоритмов обработки данных. При этом в крайних значениях рейтингов предполагается (сверху и снизу), что полностью отсутствовало проектирование подобного программного продукта или такие продукты были освоены для серийного производства.

Согласованность (жесткость) $F2$ проекта программного продукта определена как характеристика, отражающая возможность обеспечить его соответствие заданным ***требованиям и спецификациям заказчика*** к отдельным компонентам, к внутренним и внешним интерфейсам, ко всему комплексу программ в процессе его проектирования и производства. В первом предельном случае считается, что все требования должны быть выполнены обязательно полностью (***заказной продукт***), а в другом крайнем варианте рейтингов эти требования могут рассматриваться только как необязательные пожелания заказчика.

В модели COCOMO II значительное внимание уделено ***влиянию качества организации и взаимодействия коллектива квалифицированных разработчиков*** $F4$ на трудоемкость создания сложных программных продуктов (см. таблицу). В составе организационных характеристик коллектива рекомендуется учитывать: согласованность целей специалистов, участвующих в проекте, их психологическую совместимость и способность к дружной коллективной работе, наличие опыта работы в данном коллективе, а также другие объективные и субъективные свойства участников проекта. Большое значение при этом может иметь личная мотивация и психологические особенности поведения специалистов при комплексной работе над проектированием и производством сложного про-

граммного продукта. Эти характеристики обобщены в показатель ***влияния коллективности — сложности взаимодействия специалистов*** *F*4 в коллективе, которому сопоставлены рейтинги изменения трудоемкости производства продукта. Наилучшим считается непрерывное корректное взаимодействие организованной ***"команды"*** специалистов с большим опытом работы в данном коллективе при полной согласованности их целей, планов и методов работы. В остальных случаях может отсутствовать тесное взаимодействие отдельных специалистов, ввиду чего возрастает (даже в несколько раз) трудоемкость производства продукта.

В модели COCOMO II ***для прогнозирования экономических характеристик при проектировании и производстве программного продукта рекомендуется*** учитывать уровень ***методологии*** производства и обеспечения качества сложных программных комплексов на основе ***системы и модели оценки зрелости CMMI*** (*Capability Maturity Model Intergation*) [6, 7] применяемых технологических процессов. На предприятиях, достигших высокого уровня зрелости, процессы производства должны принимать статус стандарта, фиксироваться в организационных структурах и определять производственную тактику и стратегию корпоративной культуры производства и системы обеспечения качества программных продуктов. Методология CMMI рекомендует большой комплекс процессов, который предприятие должно выполнять для приобретения, поставки, производства, использования и сопровождения крупных комплексов программ, а также виды деятельности, характеризующие степень технологической зрелости этих процессов. В модели COCOMO II приводятся количественные рекомендации ***коэффициентов влияния уровней зрелости CMMI на трудоемкость*** реализации сложных программных продуктов. Применение совершенных технологий и инструментальных средств высшего уровня зрелости может снизить трудоемкость производства сложных программных продуктов более чем в пять раз по сравнению с наиболее широко применяемыми технологиями второго — третьего уровня зрелости CMMI.

Влияние характеристик специалистов, технологической и компьютерной среды на результаты прогнозирования экономических характеристик призводства программного продукта

В модели COCOMO II изложены экспертные оценки для учета влияния ***квалификации различных категорий специалистов*** на экономические показатели производства программного продукта (см. таблицу). Выбор и использование значений таких характеристик на результаты прогнозирования экономических показателей требует высокой квалификации экспертов и детального знания особенностей коллектива специалистов и среды производства реального продукта. Для обеспечения возможности их использования при прогнозировании трудоемкости и длительности разработки сложных комплексов программ в модели COCOMO II представлены соответствующие относительные рейтинги влияния основных категорий специалистов (*M*9—*M*14 — см. таблицу). При применении этих рейтингов следует учитывать, что некоторые из них взаимосвязаны. Как следствие, целесообразно анализировать их корреляцию, возможность объединения или исключения при оценке реальных проектов и коллективов специалистов.

Среди ряда ***характеристик коллектива разработчиков*** наибольшее влияние на трудоемкость оказывает ***тематическая и технологическая квалификация специалистов.*** Совместно эти факторы могут изменять трудоемкость производства программных продуктов на 30—40 %. Организационное разделение специалистов, осуществляющих ***производство программного продукта*** (первая категория), и ***специалистов-технологов***, обеспечивающих, контролирующих и управляющих его качеством в процессе производства (вторая категория), обеспечивает независимый, достоверный контроль затрат и качества результатов производства.

Специалисты первой категории непосредственно создают компоненты и комплекс программ в целом с заданными показателями качества. В процессе производства их функции заключаются в тщательном соблюдении принятой в организации-исполнителе технологии и в формировании всех исходных и отчетных документов, предписанных нормативами по соблюдению этой технологии. При этом предполагается, что выбранная технология способна обеспечить необходимые значения конструктивных показателей качества, а достижение заданных функциональных характеристик гарантируется ***тематической квалификацией*** соответствующих специалистов и регулярным контролем этих характеристик в процессе производства.

Тематическая квалификация и опыт специалистов (*M*12) в конкретной прикладной области, для которой разрабатывается программный продукт, приближенно может оцениваться продолжительностью работы по данной тематике. При низкой тематической квалификации допускаются наиболее грубые системные ошибки, требующие больших затрат при доработке программ. Представленные в разных публикациях оценки показывают, что при изменении опыта работы в данной области от 1 до 10 лет производительность труда может повышаться в 1,5—2 раза.

Технологическая квалификация (*M*13) программистов в использовании инструментальной системы автоматизации производства программных компонентов отражает опыт применения методов, средств и всего технологического процесса при создании данного типа программных комплексов. Опыт применения конкретного комплекса автоматизации, языков проектирования и программирования может являться существенным фактором при выборе технологии для создания новых компонентов и обеспечении качества программных продуктов.

Оценка производительности коллектива существенно зависит от ***стабильности состава и психологического климата*** (см. *M*11 в таблице) в коллективе специалистов и их способности к сотрудничеству и дружной совместной работе над единым продуктом. В данном факторе годы работы с конкретной технологической системой отражают не только опыт работы с инструментами, но и ***слаженность коллектива*** по проведению больших комплексных работ. Нарушение технологии, задержка при разработке отдельных модулей или групп программ могут приводить к большим дополнительным затратам и значительной задержке производства продукта в целом.

Специалисты второй категории — технологи, обслуживающие и сопровождающие технологический инструментарий, который применяется специалистами первой категории, должны обеспечивать применение системы качества предприятия, контролировать и инспектировать процессы производства для минимизации затрат. Основные задачи второй категории специалистов должны быть сосредоточены на контроле ***экономических характеристик*** процессов и результатов выполнения работ, а также на принятии организационных и технологических мер для достижения их необходимого качества, обеспечивающего выполнение всех требований технического задания на программный продукт. ***Технологи*** должны уметь выбирать, приобретать и осваивать экономически наиболее эффективный инструментарий для заказных проектов, которые реализуются конкретным предприятием, с учетом особенностей создаваемых комплексов программ требуемого качества и рентабельности технологических средств.

Инструментальные системы (*M*15), поддерживающие производство, могут быть описаны качественными характеристиками и рейтингами, изменяющими трудоемкость в пределах приблизительно 20 % от средней (номинальной). Уровень технологии и комплекса инструментальных средств особенно сильно влияет на ***экономику крупных программных продуктов***. ***Затраты на технологию и программные средства автоматизации производства*** обычно являются весьма весомыми при использовании высокоэффективных автоматизированных технологий и инструментов. При ***экономическом обосновании проекта следует учитывать***, что размер и сложность создаваемого продукта значительно влияет на выбор инструментальных средств и уровня автоматизации технологии, а также на их долю в общих затратах на производство. Возможны ситуации, при которых затраты на технологию достигают 30...50 % общих затрат на производство. Такие затраты могут быть оправданы повышением производительности труда, сокращением сроков разработки и последующим снижением затрат на множество версий программного продукта. Процесс оценки качества технологической базы производства на предприятии-исполнителе позволяет ***прогнозировать возможное качество программного продукта*** и ориентировать заказчика и пользователей при выборе разработчика и поставщика для определенного проекта с требуемыми характеристиками качества производства.

В некоторых комплексах программ ***реального времени*** на увеличение трудоемкости разработки (до 30...50 %) может оказать влияние ***ограничение вычислительных ресурсов*** оперативной памяти (*M*7) и производительности объектного компьютера (*M*6), на которых должен функционировать программный продукт (бортовые системы). Это обстоятельство привело к необходимости разработки ***методов эффективного использования программами аппаратных ресурсов компьютеров***. Влияние ***ограничения объема памяти реализующего*** компьютера (*M*7 в модели COCOMO II) определяется долей от фактического размера доступной оперативной памяти, которая может использоваться для размещения программного продукта и данных при решении возлагаемых на компьютер совокупности функций с приемлемой для практики точностью и надежностью. На практике наибольшие технические трудности обычно вызывают ***ограничения производительности компьютера*** (*M*6), ***реализующего заказной комплекс программ.*** Для их преодоления приходится не столько экономить программные ресурсы и сокращать размеры программного продукта, сколько искать пути увеличения производительности компьютерных систем и эффективности их использования. Существенно ограничивающим ***фактором при этом является реальное время***, в течение которого компьютерные ресурсы могут быть предоставлены для решения данной задачи, или то время, в пределах которого целесообразно получить результаты (время отклика) для их практического использования.

Быстрый рост числа решаемых задач, их сложности и требуемой производительности вычислительных средств стимулирует поиск путей удовлетворения потребностей заказных программных продуктов в ресурсах для решения таких задач. В последние годы стали возможными ситуации, когда ***в современных компьютерных системах реального времени*** может быть создан такой избыток производительности и памяти, что нет необходимости в детальном выборе и применении методов рационального и экономного использования вычислительных ресурсов при экономическом прогнозировании и обосновании таких проектов. При ограниченных вычислительных ресурсах вследствие требований минимизации весов и габаритов специализированных компьютеров в авиационных, ракетных и космических системах их ***экономное использование остается актуальным***, и это следует учитывать при производстве соответствующих программных продуктов.

Примеры прогнозирования экономических характеристик производства программных продуктов

Ниже представлены и анализируются ***гипотетические варианты прогнозов экономических характеристик программных продуктов***, рассчитанные с использованием программного пакета ПЛАПС-2 — разработчик Н. Липкин. Пакет реализует алгоритмы прогнозирования по формулам модели COCOMO II (выражения (1) и (2)) и предназначен для прогнозирования экономических характеристик (подробнее см. описание и рекламный листок ПЛАПС-2 в работе [4]). При использовании пакета возможно указание различных факторов и характеристик проекта, а результаты отображаются сверху в окне ***"Экономические характеристики"***. В них содержатся значения прогнозов трудоемкости и длительности реализации проекта, а также оценки производительности труда и числа специалистов, необходимых для производства целевого комплекса программ. Параметры модели логически сгруппированы и расположены на различных вкладках в соответствии со структурой таблицы.

Представленные ***примеры вариантов*** могут использоваться как ***иллюстрации и ориентиры*** при прогнозах необходимых затрат на крупные программные продукты. Модель COCOMO II позволяет определять и уточнять некоторые исходные данные для ***экономического прогнозирования производства конкретного программного продукта***. При этом, как правило, значения ряда факторов являются фиксированными в силу объективных условий производства на конкретном предприятии. В отдельных случаях для выбора наилучшей стратегии реализации проекта может быть полезным варьирование нескольких вариантов характеристик факторов перед детальным проектированием производства конкретного комплекса программ. В крайнем случае, на основе выполненных оценок на этапах предварительного или детального проектирования может быть сделано достаточно достоверное ***обоснование прекращения разработки*** предполагавшегося программного продукта.

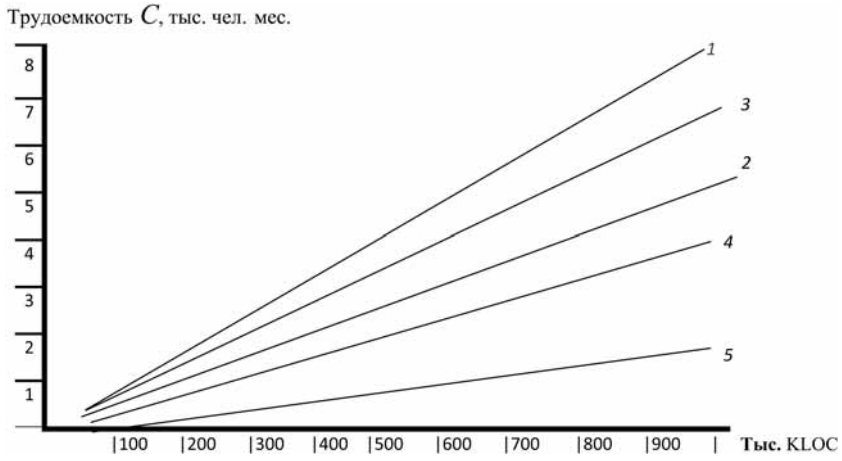


Рис. 1. Зависимость трудоемкости разработки от размера программного продукта

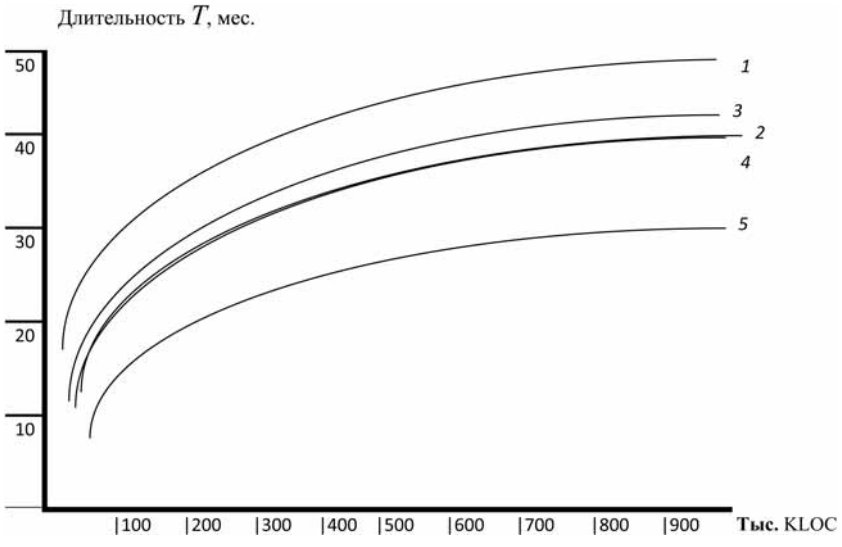


Рис. 2. Зависимость длительности разработки от размера программного продукта

Ниже анализируются ***программные продукты*** размером от 50 до 1000 слов KLOC (миллион строк программ). Состав факторов детальной модели COCOMO II анализируемых программных продуктов представлен в таблице. Для каждого варианта в зависимости от размеров прогнозируемых комплексов программ представлены графики трудоемкости *C* (рис. 1), длительности производства *T* (рис. 2) и числа *N* необходимых специалистов (рис. 3). Первые два варианта (*1*, *2*) иллюстрируют влияние на прогнозируемые трудоемкость, длительность и число специалистов уровня зрелости CMMI (2 или 4) и организации коллектива (свободная или жестко планируемая), которые повышаются с ростом квалификации, слаженности коллектива специалистов и оснащенности технологией и инструментальными средствами. Варианты *3—5* отражают различную долю повторного использования компонентов в высоконадежных программных продуктах реального времени при ограниченных ресурсах компьютеров (бортовые системы).

Вариант 1 отличается тем, что должен быть создан совершенно новый программный продукт средней сложности и надежности, однако коллектив имеет невысокую квалификацию и слаженность, относительно слабо оснащен технологией и инструментарием (уровень 2 CMMI).

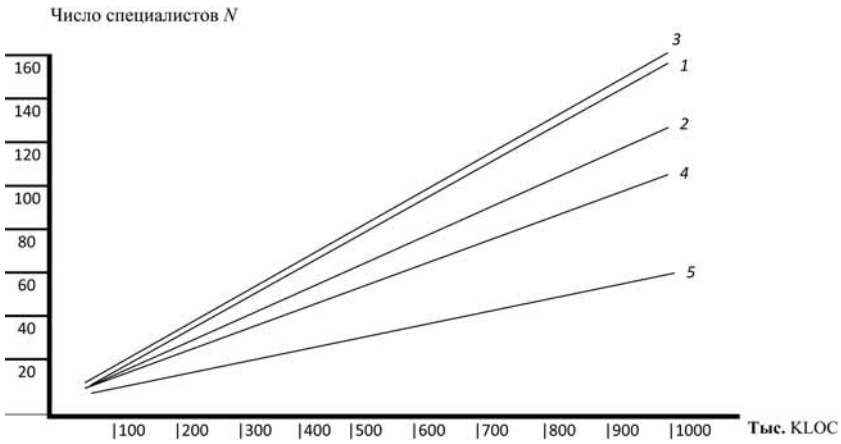


Рис. 3. Зависимость числа специалистов от размера программного продукта

В результате экономические характеристики близки к наиболее низким по производительности труда, требуется наибольшая трудоемкость и число специалистов.

Вариант 2 — предполагается, что должен создаваться почти такой же как в предыдущем варианте, совершенно новый программный продукт высокой сложности и надежности в соответствии с жесткими требованиями заказчика. Разработка продукта предстоит стабильному коллективу, включающему специалистов высокой квалификации и слаженности, имеющему значительный опыт создания подобных проектов. Процесс производства поддержан современным инструментарием и технологией на уровне 4 CMMI. Остальные факторы имеют, в основном, номинальные значения. Этот вариант может считаться лучшим по трудоемкости и производительности труда и требует меньшего числа специалистов по сравнению с предыдущим вариантом. Трудоемкость при этом сокращается в полтора раза, а длительность производства программного продукта только на 15 %.

Вариант 3 должен быть создан в соответствии с требованиями заказчика на не совсем новый программный продукт реального времени средней сложности и надежности, компоненты которого допускают повторное использование в аналогичных проектах и их версиях. Производство продукта предстоит коллективу специалистов средней квалификации и слаженности, имеющему опыт создания подобных проектов. Процесс производства поддержан современным инструментарием и технологией на уровне 3 CMMI. При надежном функционировании программного продукта в реальном времени должны эффективно использоваться вычислительные ресурсы реализующего вычислительный процесс компьютера на уровне 90 % по оперативной памяти и производительности.

Вариант 4 в основном подобен по характеристикам факторов варианту *3*, но базируется на применении 50 % готовых программных компонентов, что отразилось на значительном изменении экономических характеристик. В большей степени это отразилось на снижении трудоемкости и числа специалистов (почти в два раза), и относительно меньше (только на 10 %) сократились длительность производства и производительность труда.

Вариант 5 отличается от предыдущего более высоким уровнем повторного использования компонентов — 80 %. Это отразилось на значительном снижении трудоемкости и необходимого числа специалистов (на 70 %), а также на повышении производительности труда при относительно небольшом сокращении длительности производства (только на 30 %).

Характерными особенностями представленных графиков *2* и *4* является практически линейная зависимость трудоемкости производства и числа необходимых специалистов от предполагаемого размера программного продукта. Однако длительность производства комплексов программ (см. рис. 2) отражается нелинейными кривыми с постепенным ограничением их величины.

Представленные прогнозы могут служить ***ориентирами*** при анализе реальных условий производства программных продуктов. Уменьшение затрат труда в последние годы объективно обусловлено усовершенствованием технологий и инструментария при производстве программных продуктов. Средняя производительность труда для новых относительно небольших комплексов программ при высоком качестве технологий повышается более чем в два раза, а затем несколько сокращается по мере учета возрастания их размера.

При ***обобщении рассчитанных экономических характеристик*** на практике некоторые значения могут не удовлетворить специалистов, ведущих прогнозирование. По этой причине в методиках прогнозирования должна допускаться ***возможность пересчета*** получаемых прогнозных значений на основе уточненных значений некоторых факторов. Например, на базе учтенных при прогнозе факторов производительность труда может оказаться отличающейся от реальной, характерной для предприятия-исполнителя, определявшейся по реализованным предшествующим разработкам. Для того, чтобы учесть это различие, должна быть предусмотрена возможность ввода реальной производительности специалистов данного предприятия и пересчета всех остальных экономических характеристик. При этом индивидуальная производительность труда отдельных специалистов коллектива может существенно отличаться от "усредненной", заложенной в модель по статистике предыдущих разработок различного размера. Кроме того, на практике могут существовать ограничения реальной численности коллектива, выделяемого для данного проекта комплекса программ, или допустимой (директивной) длительности производства. Для учета таких ограничений должна быть предусмотрена возможность пересчета экономических характеристик при дополнительно откорректированных значениях некоторых факторов.

Достоверность рассчитанных значений основных экономических характеристик целесообразно проверять путем сопоставления с показателями аналогов, созданных на том же предприятии, наиболее близких к прогнозируемому продукту. Полная стоимость и длительность разработки обычно подлежат согласованию с заказчиком. В процессе согласования уточняются сценарии проектирования и производства и, возможно, изменение не только влияния некоторых факторов, но и требований технического задания на объект и характеристики продукта. Такие действия особенно необходимы, если превышаются допустимые заказчиком значения стоимости или длительности разработки.

Согласованные с заказчиком прогнозируемые экономические характеристики позволяют фиксировать базовый сценарий и план производства продукта, проводить предварительные расчеты распределений трудоемкости и длительности по этапам производства и по специалистам. По этим данным может проводиться детальное календарное планирование и контролирование работ всего проекта. Для расчета распределения трудоемкости, длительности и числа специалистов по этапам работ желательно использовать аналогичные характеристики проектов подобного прототипа. Представленные выражения, таблицы факторов и рейтингов целесообразно использовать ***как ориентиры и методическую базу для прогнозирования экономических характеристик*** производства сложных программных продуктов, которые необходимо уточнять и развивать на основе анализа реальных характеристик и опыта выполненных проектов программных продуктов.

Заключение

Возрастание ***сложности и ответственности современных задач***, решаемых сложными заказными системами, а также возможного ущерба от недостаточного качества программных продуктов значительно повысило актуальность освоения методов стандартизированного описания требований прогнозирования экономических характеристик и качества на различных этапах их жизненного цикла. Выявилась необходимость систематизации реальных характеристик качества программных продуктов, применения стандартов для выбора из них необходимой номенклатуры и требуемых решений для конкретных комплексов программ. Тем самым должен быть выделен важный, базовый раздел из всей экономики жизненного цикла программных комплексов. Внимание должно быть сосредоточено на концептуальной основе распределения затрат труда в процессе разработки компонентов и комплексов программ, на ***факторах, определяющих реальные трудозатраты и другие экономические характеристики***, а также на исследовании их в реализованных современных разработках.

Для каждого крупного проекта, выполняющего ответственные функции, необходимо ***прогнозировать требующиеся ресурсы***, разрабатывать и применять комплексную систему качества, специальные планы и программу, методологию и инструментальные средства, обеспечивающие требуемые качество, надежность и безопасность функционирования программного продукта. Для этого следует применять современные методы и стандарты при подготовке промышленных технологий, методик производства и испытания конкретных продуктов, однозначно отражающих степень удовлетворения исходных требований заказчика и пользователей, а также для сравнения продуктов разных поставщиков и выявления среди них предпочтительных.

Обещания разработчиков в контрактах с заказчиками создать высококачественные продукты в согласованные сроки во многих случаях не выполняются как ввиду различий в понимании требуемого качества, так и по причине неумения корректно прогнозировать экономические ресурсы, необходимые для достижения заданного качества программ. Стратегической задачей в жизненном цикле современных систем стало ***прогнозирование, обеспечение и совершенствование качества производства сложных программных продуктов при реальных ограничениях на использование доступных экономических ресурсов***.

В жизненном цикле сложных комплексов программ для обеспечения их высокого качества ***целесообразно выделять специалистов, ответственных за анализ, оценивание и прогнозирование экономических характеристик производства***, за соблюдение промышленной технологии создания и совершенствование программных продуктов, за измерение и контроль затрат, качества комплексов программ в целом и их компонентов. ***Задача состоит в том, чтобы научить и приучить специалистов*** к анализу и оцениванию конкретных экономических факторов, влияющих на характеристики функционирования программных продуктов со стороны реально существующих и потенциально возможных ограничений ресурсов проектов. Необходима подготовка и воспитание квалифицированных специалистов ***в области индустрии, экономики и производства сложных программных комплексов***, их обучение методам и современной программистской культуре промышленного создания крупных высококачественных программных продуктов.

Список литературы

1. **Боэм Б. У.** Инженерное проектирование программного обеспечения / Пер. с англ. под ред. А. А. Красилова. М.: Радио и связь, 1985. (Boehm B. W. Software Engineering Economics. Prentice-Hall, 1981).
2. **Boehm B. W.** et al. Software cost estimation with COCOMO II. Prentice Hall PTR. New Jersey. 2000.
3. **Липаев В. В., Потапов А. И.** Оценка затрат на разработку программных средств. М.: Финансы и статистика, 1988.
4. **Липаев В. В.** Экономика производства сложных программных продуктов. М.: СИНТЕГ, 2008.
5. **Londeix B.** Cost estimation for software development. Cornwall: Addison-Wesley, 1987.
6. **Соммервилл И.** Инженерия программного обеспечения. 6-е издание: пер. с англ. М.: Вильямс, 2002.
7. **Фатрелл Р. Т., Шафер Д. Ф., Шафер Л. И.** Управление программными проектами: достижение оптимального качества при минимальных затратах: пер. с англ. М.: Вильямс, 2003.

УДК 004.414.2

С. А. Букашкин, д-р техн. наук, проф., директор ФГУП "НИИ автоматики", г. Москва,
Б. Н. Кривошеин, директор департамента Радиоэлектронной аппаратуры ЗАО "Ланит-Терком", e-mail: Boris.Krivoshein@lanit-tercom.com,
А. Н. Терехов, д-р физ.-мат. наук, проф., зав. каф. СПбГУ, ген. директор ЗАО "Ланит-Терком", e-mail: Andrey.Terekhov@lanit-tercom.com,
Н. Ф. Фоминых, канд. физ.-мат. наук., главный конструктор программного обеспечения ЗАО "Ланит-Терком", г. Санкт-Петербург

Проектирование отказоустойчивого вычислительного комплекса с архитектурой 2 из 3 (2oo3)

Рассматриваются практические вопросы проектирования отказоустойчивых систем с аппаратной избыточностью на примере вычислительного комплекса ОВК с архитектурой 2 из 3. Приведены примеры оценки уровня функциональной безопасности, расчета интенсивности отказов, решения общих вопросов проектирования отказоустойчивых программно-аппаратных систем. Применение принципов проектирования отказоустойчивой аппаратуры продемонстрировано детальными описаниями алгоритмов, протоколов и других технических решений, используемых в ОВК.

***Ключевые слова**: отказоустойчивый вычислитель, мажорирование, тройная модульная избыточность, архитектура 2 из 3, функциональная безопасность, надежность*

Введение

Подходы к построению систем, гарантирующих функциональную безопасность, подробно исследованы и изложены в серии международных стандартов IEC 61508 "Функциональная безопасность систем электрических, электронных, программируемых электронных, связанных с безопасностью". В России адаптированный вариант IEC 61508 был принят как национальный стандарт ГОСТ Р МЭК 61508 [1]. Однако разнообразие реальных систем управления и обработки данных очень велико и не всегда соответствует традиционной модели "датчики — контроллер — схема защиты". Современные технологии передачи данных позволяют строить распределенные системы со сложной сетевой топологией на базе компьютеров и оконечных устройств с сетевыми интерфейсами. Для таких систем требуется специальная интерпретация положений стандарта [1] для обеспечения требуемого уровня надежности.

В данной работе мы будем рассматривать сетевую распределенную систему, которая обладает следующими особенностями:

- программируемые электронные устройства системы связаны между собой, а также с устройствами ввода и вывода посредством одного или нескольких высокоскоростных каналов пакетной передачи данных;
- каждое электронное устройство системы в непрерывном режиме выполняет функции приема, обработки и передачи данных по сети Ethernet во взаимодействии с другими сетевыми устройствами;
- система имеет высокие требования по уровню производительности, что приводит к применению сложных инженерно-технических решений при проектировании ее узлов;

• система в целом реализует функции безопасности, и любой сбой в процессах обработки и передачи данных от источника до конечного получателя является потенциально опасным событием.

Оценка полноты безопасности зависит от нескольких характеристик, таких как полнота диагностического покрытия, интервал между контрольными испытаниями, среднее время ремонта, доля отказов с общей причиной. Отказ с общей причиной определен в стандарте [1] (часть 4, п. 3.6.10) как "отказ, который является результатом одного или нескольких событий, вызвавших одновременные отказы двух и более отдельных каналов в многоканальной системе, ведущие к отказу системы". В этой же части стандарта можно найти формальные определения всех терминов, используемых в других частях стандарта [1].

Опасным отказом на уровне системы в этом случае следует считать любое из двух следующих событий:

• получение оконечным устройством недостоверных результатов обработки, таких, что ошибка не будет обнаружена средствами внутренней диагностики;

• неполучение оконечным устройством достоверных результатов обработки в течение заданного времени.

Проектирование аппаратуры с высокими требованиями по функциональной безопасности требует не только знания требований стандартов, но и владения практическими приемами разработки таких систем. В данной статье показано, как принципы отказоустойчивости могут применяться в опытно-конструкторских разработках. В качестве примера рассмотрен проект по разработке отказоустойчивого вычислительного комплекса (ОВК), выполненный совместно ФГУП "НИИ автоматики" и ЗАО "Ланит-Терком" в 2008—2011 гг.

ОВК является частью распределенной системы обработки данных, построенной на основе компьютерной сети Ethernet и обслуживающей критичные с точки зрения безопасности приложения. Взаимодействие ОВК с периферийным оборудованием осуществляется по восьми независимым интерфейсам Fast Ethernet, никаких других каналов ввода-вывода информации ОВК не имеет. Кроме сетевых интерфейсов, на корпусе ОВК расположены три разъема для подключения источников питания 24 В постоянного тока и порт защиты от несанкционированного вскрытия корпуса (рис. 1, см. вторую сторону обложки).

Структура ОВК

В состав ОВК входит 14 электронных модулей форм-фактора 6U (233 × 160 мм), объединенных внутренними высокоскоростными каналами передачи данных (рис. 2, см. вторую сторону обложки).

Распределенная вычислительная сеть может содержать одно либо несколько устройств ОВК и множество периферийных устройств. Эти периферийные устройства по сетевым интерфейсам передают исходные данные в ОВК и принимают от ОВК управляющую информацию. На уровне сети все периферийные устройства и каналы передачи данных резервированы, при этом мониторинг всей системы и управление сетевой конфигурацией выполняется на уровне ПО ОВК. Для данной структуры обеспечение бесперебойной и достоверной работы ОВК является основой функциональной безопасности системы в целом. В качестве базовой архитектуры ОВК была выбрана схема 2oo3 (запись *M*oo*N* означает, что в составе устройства имеется *N* каналов, и для нормального функционирования устройства достаточно корректной работы *M* из них, запись *M*oo*ND* используется для архитектуры *M*oo*N* со встроенной диагностикой). Данная архитектура состоит из трех каналов, соединенных параллельно, с мажорированием выходных сигналов так, что выходное состояние не меняется, если результат, выдаваемый одним из каналов, отличается от результата, выдаваемого двумя другими каналами ([1], часть 6, В.2.2.5). Для исключения возможных систематических ошибок ПО все функции контроля и управления избыточными ресурсами были вынесены на аппаратный уровень.

При строгом следовании базовым принципам построения систем с аппаратной избыточностью, изложенным в работе [2], архитектура 2oo3 обеспечивает уровни полноты безопасности, удовлетворяющие требованиям SIL3/SIL4 стандарта IEC 61508 "Функциональная безопасность систем электрических, электронных, программируемых электронных, связанных с безопасностью" ([1], часть 2). Все программное обеспечение ОВК исполняется в синхронном режиме на трех независимых комплектах оборудования, однако этот уровень скрыт для пользователя. С точки зрения разработчика прикладного ПО, ОВК представляет собой одноядерный компьютер с восьмью сетевыми интерфейсами Fast Ethernet и встроенными средствами самодиагностики. Целевая вероятность опасного отказа для аппаратуры ОВК $PFH_{\text{ОВК}}$ задана на уровне не более 10^{-8}, что позволяет отнести требуемый уровень полноты безопасности ОВК к категории SIL4 без учета ПО.

Периферийные устройства сети под управлением ОВК используют для передачи данных каналы связи с относительно низким уровнем надежности. Модель применения ОВК предполагает обеспечение защиты от сбоев в каналах передачи данных программным путем на уровне приложения. Для диагностики сбоев на уровне каналов связи функциональное ПО ОВК должно содержать модуль с функцией программного узла защиты от ошибок (УЗО, рис. 3). Кроме того, защита от потери либо искажения пакетов данных выполняется на уровне стека протоколов TCP/IP. Избыточное кодирование Ethernet-пакетов гарантирует вероятность пропуска ошибки при приеме искаженного пакета не более 2^{-32}. Для максимальной интенсивности обмена по каналу связи со скоростью 100 Мбит/с и вероятности битовой ошибки при передаче, равной $1 \cdot 10^{-12}$, типичной для каналов Ethernet, интенсивность отказов схемы защиты канала λ составляет

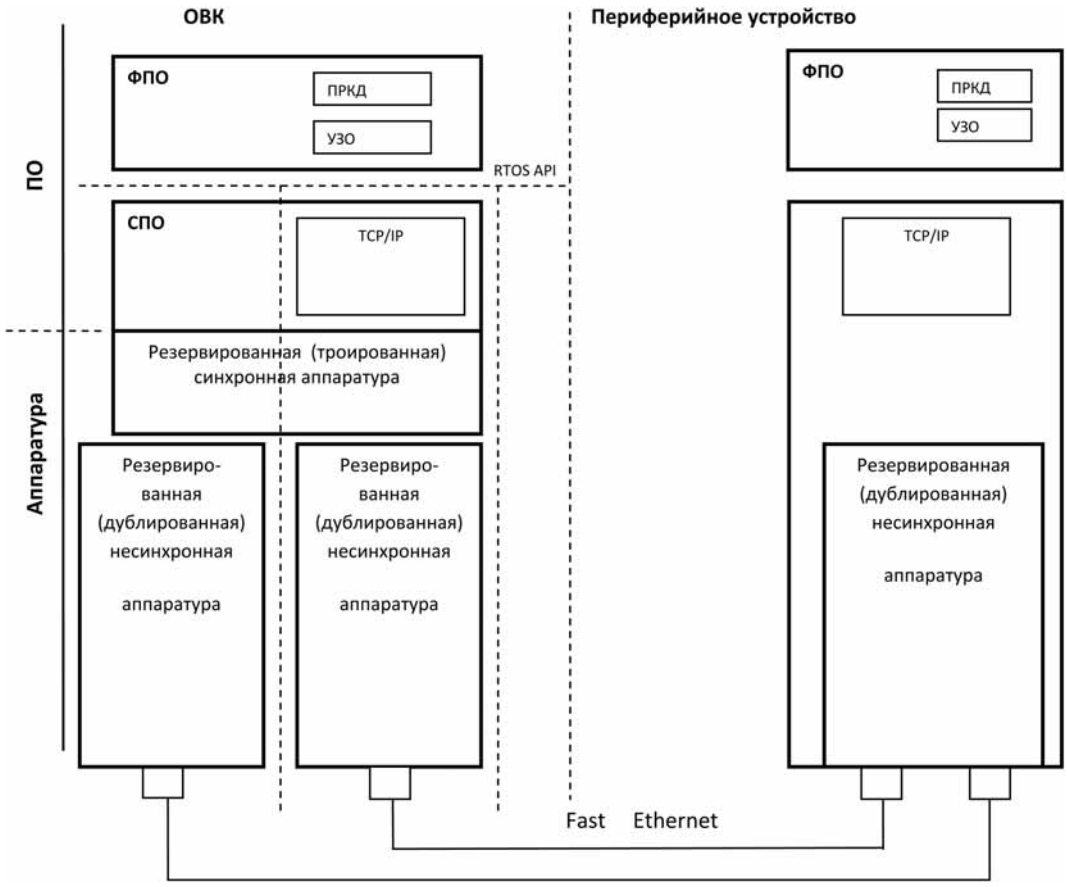


Рис. 3. Модель взаимодействия ОВК с периферийным устройством:
ФПО — функциональное программное обеспечение; СПО — системное программное обеспечение; ПРКД — прикладное программное обеспечение; RTOS API — Real Time Operating System Application Program Interface

$8{,}4 \cdot 10^{-11}$, что является достаточным уровнем для исключения опасных отказов по этой причине.

Высокая надежность управления в масштабе сети обеспечивается выполнением ФПО и УЗО средствами синхронно работающей аппаратуры с архитектурой 2oo3. Системная программная платформа, создающая среду выполнения прикладного ПО, работает на этом же уровне архитектуры и защищена от случайных ошибок аппаратных средств. В то же время аппаратные компоненты ОВК, реализующие канал передачи данных, могут иметь более высокую интенсивность отказов, типичную для нерезервированных систем. Возможные ошибки в этой части будут обнаружены УЗО и обработаны как сбои в канале связи, без распространения последствий таких ошибок на уровень ФПО. Данная структура позволяет свободно перераспределять сетевую функциональность платформы между программными и аппаратными, синхронными и несинхронными, резервированными и нерезервированными компонентами ОВК без возможных потерь с точки зрения обеспечения функциональной безопасности.

Внутренняя структура ОВК определяется требованиями по надежности и моделью использования в составе сети. Вычислительное ядро, на котором выполняется все программное обеспечение, должно быть полностью реализовано в рамках архитектуры 2oo3. Порты каналов Ethernet образуют уровень взаимодействия с периферией по интерфейсам без архитектурной избыточности. Для придания системе отказоустойчивости по отношению к отказам каналов связи могут быть использованы сетевые технологии объединения нескольких одиночных интерфейсов в логические каналы. Технология агрегации каналов (*Link aggregation* [3]) в случае отказа физического интерфейса, входящего в состав логического канала, обеспечивает передачу данных по работающим каналам без прерывания сервиса. С точки зрения архитектуры системы, это соответствует схеме 1oo$N$, где N — число физических интерфейсов, составляющих объединенный логический канал. Данная модель не требует каких-либо аппаратных решений для поддержки архитектуры 1oo$N$ на уровне канала связи, однако в ОВК необходимо согласование двух архитектурных решений — внутреннего 2oo3 и внешнего 1oo$N$. Кроме того, для объединения трех комплектов синхронной аппаратуры по схеме 2oo3 требуются мажорирующие элементы, которые в целях защиты от отказов по общей причине должны быть реализованы отдельно от вычислителей. Таким образом, между уровнем вычислителей и уровнем физических интерфейсов должен

находиться третий, согласующий уровень аппаратуры, выполняющий следующие основные функции:

- прием пакетов данных с уровня физических интерфейсов и их синхронная передача по трем каналам на уровень вычислителей;
- синхронный прием пакетов данных от вычислителей, их мажорирование и отправка на уровень физических интерфейсов.

Со стороны вычислителей уровень согласования дополняет решение до законченной реализации модели 2oo3, поэтому аппаратура согласования должна иметь аналогичные вычислителям характеристики надежности. Естественным решением является построение согласующей аппаратуры по той же модели 2oo3. Соответствующая структура ОВК приведена на рис. 4 (см. вторую сторону обложки).

С точки зрения архитектуры, три модуля центрального процессора CPU образуют уровень вычислителей, три модуля ввода-вывода I/O — уровень согласования аппаратуры вычислителей и интерфейсных модулей и восемь модулей Ethernet — собственно интерфейсный уровень, обеспечивающий сетевое подключение ОВК по восьми физическим портам. Схема соединений между внутренними уровнями ОВК обеспечивает полный контроль операций согласующего уровня и его соответствие архитектурной схеме 2oo3.

Схема питания является отдельным архитектурным элементом. Контроль электропитания осуществляется независимо каждым функциональным блоком, при этом для нормальной работы достаточно работы любой из трех входных линий питания. Таким образом, архитектура подсистемы электропитания ОВК — 1oo3. Аппаратура системы электропитания ОВК не содержит активных компонентов и состоит только из разъемов для подключения трех независимых внешних источников питания и электрических соединительных линий от каждого разъема до входов питания функциональных блоков.

Оценка надежности подсистемы электропитания может быть получена по методике, описанной в стандарте MIL-HDBK-217F "Прогнозирование надежности электронного оборудования" ([4], разд. 15.1). Для каждой из трех подсистем электропитания оценка интенсивности отказов $\lambda = 3 \cdot 10^{-8}$ (при расчете использовались параметры: рабочая температура $t = 40$ °C, интенсивность циклов подключения-отключения разъемов не более 0,5 на 1000 ч эксплуатации). Это достаточно высокий показатель надежности, не приводящий к необходимости тройного резервирования для всей подсистемы питания, однако выбранная архитектура 1oo3 позволяет снизить требования по надежности к внешним источникам питания ОВК.

Расчет надежности

Для расчета показателей надежности ОВК в целом для конфигурации с четырьмя дублированными портами Fast Ethernet воспользуемся следующей схемой, представленной на рис. 5.

По методике, определенной в стандарте [1] (часть 6, прил. В), с допущениями, необходимыми для использования формулы, приведенной в разделе "Приближенные формулы для определения интенсивности отказов восстанавливаемых параллельных системы" (табл. В.2 стандарта [5]), получим следующую оценку вероятности отказа аппаратных средств ОВК:

$$PHF_{\text{OBK}} = \left(6\lambda_{\text{PW}}^3 + 6\lambda_{\text{CPU}}^2 + 6\lambda_{\text{IO}}^2 + 4 \cdot 2\lambda_{\text{FE}}^2\right) MTTR \approx$$
$$\approx 3{,}72 \cdot 10^{-10} \cdot 8 \approx 3 \cdot 10^{-9}. \quad (1)$$

При заданной целевой вероятности отказов $PFH_{\text{OBK}} \leqslant 1 \cdot 10^{-8}$ данная оценка позволяет снизить требования к среднему времени восстановления в процессе эксплуатации *MTTR* до 24 ч.

Относительно высокие параметры надежности модулей CPU, I/O и FE, использованные для данной оценки, были достигнуты за счет реализации всех логических и диагностических функций каждого из модулей на единственной ПЛИС (программируемой логической интегральной схеме) Virtex-5 компании Xilinx. Данная серия ПЛИС обладает исключительно высокой надежностью, которая подтверждается "Отчетом о надежности аппаратуры" [6], регулярно пуб-

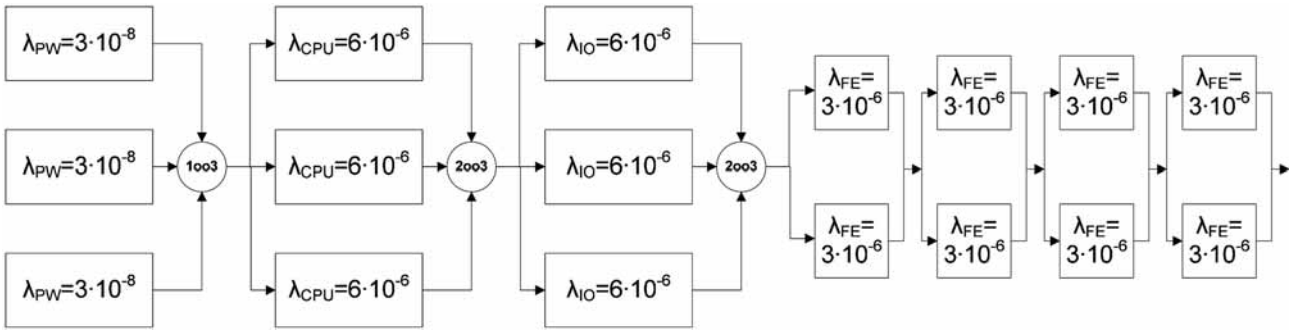


Рис. 5. Схема расчета надежности ОВК:

λ_{PW} — интенсивность отказов одного канала электропитания; λ_{CPU} — интенсивность отказов одного модуля CPU; λ_{IO} — интенсивность отказов одного модуля I/O; λ_{FE} — интенсивность отказов одного модуля Fast Ethernet

ликуемым производителем. По данным отчета, для микросхем семейства Virtex-5 интенсивность отказов λ составляет $6 \cdot 10^{-9}$ во всем допустимом температурном диапазоне, определенном спецификацией. ПЛИС Virtex-5 оснащена большим числом высокоскоростных интерфейсов со встроенными контроллерами физического уровня, что позволяет свести к минимуму число компонентов на модуле и тем самым увеличить его надежность.

Приведенная оценка вероятности отказа ОВК является обоснованной только при том условии, что при проектировании были соблюдены все допущения, которые позволяют перейти от полной формулы расчета вероятности отказа системы с архитектурой 2oo3 ([1], часть 6, В.3.2.5) к упрощенной (1), учитывающей только интенсивности отказов элементов и среднее время восстановления. К этим допущениям относятся следующие:

- Исключение отказов по общей причине. Как следствие этого требования, система не должна иметь элементов, отказ которых может повлечь за собой ошибку в работе более чем одного из модулей. Кроме того, на уровне информационного взаимодействия все модули должны принимать любые возможные состояния соседних модулей и каналов связи как допустимые и корректно их обрабатывать. Для поддержания эквивалентности логических состояний синхронно работающих модулей любое событие в системе, требующее обработки на уровне ПО, должно гарантированно и синхронно доставляться трем вычислителям.
- Интервал времени между процедурами тестирования принят равным нулю. Для выполнения этого условия в системе должны быть постоянно активными процессы контроля внутренних операций и фонового тестирования.
- Одиночные ошибки вычислений и данных в памяти корректируются автоматически, при этом время восстановления системы после сбоя настолько меньше среднего времени восстановления после постоянного отказа, что этой составляющей можно пренебречь. Причинами одиночных ошибок могут быть электромагнитные помехи, взаимодействие полупроводников с альфа-частицами, джиттер синхронизирующих сигналов и др. Подобные события не требуют ремонта аппаратуры для восстановления нормальной работы, однако логические состояния модулей могут разойтись, и с точки зрения функциональной безопасности такое событие эквивалентно отказу элемента, перешедшего в несинхронное состояние.

При проектировании системы, соответствующей данным требованиям, необходимы технические решения, обеспечивающие синхронную независимую работу всех модулей, диагностику их состояний и восстановление единого логического состояния после возможного сбоя.

Синхронизация

ОВК не может иметь единого источника синхронизации, так как он являлся бы общей точкой отказа. При начальном старте системы каждый из модулей независимо выполняет процедуру инициализации своих локальных ресурсов и затем переходит в режим установления логической синхронизации с другими модулями. Алгоритм входа в синхронизацию основан на обмене синхроимпульсами начала временного цикла обмена. В начальном состоянии каждый из модулей генерирует импульсы синхронизации с заданной частотой и одновременно принимает такие же импульсы от остальных модулей. Целью процесса начальной синхронизации является выравнивание фаз генераторов синхроимпульсов. Если модуль не синхронизован ни одним из двух оставшихся, прием импульса синхронизации приведет к немедленному изменению фазы собственного генератора и переходу в синхронное состояние. В синхронном состоянии возможен обмен информацией между модулями, при этом двух модулей достаточно для нормальной работы. Переход в синхронное состояние последнего модуля выполняется им автономно, тем же методом коррекции фазы собственного генератора синхроимпульсов. Два находящихся в синхронном состоянии модуля нечувствительны к синхроимпульсам третьего, и это позволяет ОВК продолжать работу при любом характере неисправности одного из модулей.

Выравнивание импульсов синхронизации между модулями не является абсолютно точным, но этого достаточно для корректной работы системы. Каждый импульс синхронизации генерируется на основе частоты локального осциллятора, при этом фазы осцилляторов разных модулей могут быть произвольными, а частоты — изменяться в определенных пределах. На практике разность частот может привести к тому, что временное смещение между импульсами синхронизации, генерируемыми разными модулями, будет увеличиваться и выйдет за рамки диапазона, допустимого для модели синхронного обмена. Для поддержания системы в синхронном состоянии, схемы контроля синхронизации постоянно анализируют разность фазы собственного и внешних генераторов синхроимпульсов и, в случае необходимости, выполняют коррекцию своей фазы.

После перехода модулей процессоров в синхронное состояние система готова к выполнению контролируемых вычислений в соответствии с архитектурной моделью 2oo3. ПО вычислительных ядер всех модулей CPU идентично и выполняется логически синхронно. Под "логической синхронностью" понимается полное совпадение состояний вычислительных ядер на N-ом такте вычислений для любого N.

Логическая синхронность процессоров ОВК обеспечивается:

- управлением тактовой частотой, на которой работает вычислительное ядро. Временные циклы обмена информацией между модулями одновременно являются вычислительными циклами, на каждом из ко-

торых выполняется одинаковое, заранее определенное число операций в каждом из процессоров. После завершения очередного вычислительного цикла и до начала следующего тактовые частоты процессоров приостанавливаются. Таким образом, логическая синхронность модулей сохраняется постоянно, несмотря на взаимные отклонения локальных тактовых частот;

- контролем доставки без искажений данных, имеющих одиночный источник, всем вычислителям ОВК. Выполнение синхронных операций над различными наборами данных неизбежно приведет к потере логической синхронности модулями CPU, что является одним из сценариев отказа по общей причине;
- привязкой к номеру такта любых асинхронных событий, включая прерывания. При возникновении в ОВК любого события, внешнего по отношению к процессу синхронных вычислений, информация об этом событии сохраняется в модулях I/O и передается модулям CPU одновременно в следующем за возникновением события вычислительном цикле;
- использованием только синхронных интерфейсов процессора с жестко детерминированными протоколами обмена и исключением блокирующих событий. Любой обмен данными с переменным временем выполнения может привести к потере логической синхронности модулей CPU и поэтому запрещен.

Под управлением ПО вычислительное ядро модулей CPU синхронно взаимодействует с локальными ресурсами, включая контроллер синхронизации (SC), и модулями ввода-вывода I/O.

Интерфейс межмодульного взаимодействия ОВК реализован на основе протокола нижнего уровня Aurora. Это высокоэффективный протокол с малым временем ожидания, позволяющий передавать данные между модулями по последовательному высокоскоростному интерфейсу. Каждый из трех модулей CPU соединен с каждым из трех модулей I/O отдельным каналом связи с топологией "точка—точка" (рис. 6), что обеспечивает независимость процессов обмена данными и исключает конфликты при доступе к среде передачи. Каждый межмодульный канал Aurora объединяет четыре последовательных интерфейса RocketIO с полосой пропускания 3 Гб/с, что соответствует общей производительности канала передачи данных для каждой пары взаимодействующих модулей 12 Гб/с в режиме полного дуплекса.

Логический протокол обмена данными по межмодульным интерфейсам ОВК построен по принципу жесткого временного разделения доступа внутренних функциональных блоков к среде передачи данных. Для каждого из блоков, участвующих в межмодульных обменах, выделен определенный временной интервал внутри цикла обмена — "тайм-слот". Данная схема исключает конфликты доступа функциональных блоков к общим ресурсам и нивелирует вероятностный характер процессов обработки, свойственный программным системам. Функциональные блоки, участвующие в обмене данными, спроектированы

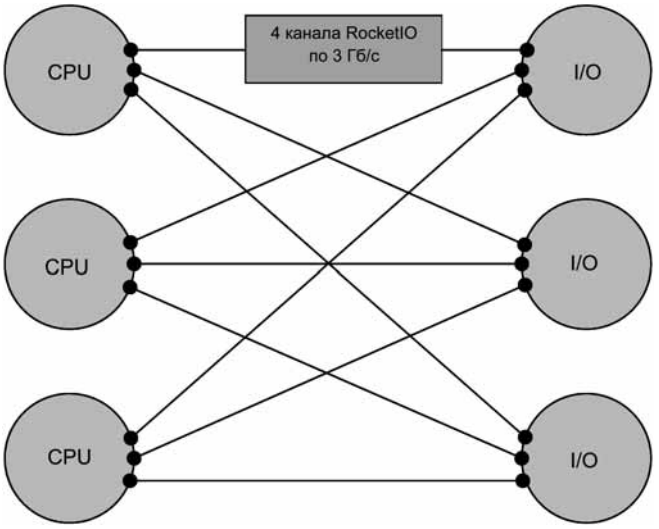


Рис. 6. Структура интерфейсов межмодульного взаимодействия ОВК

таким образом, что время передачи всего объема информации для передачи, накопленного блоком за время одного цикла вычислений, не может превышать объема выделенного ему тайм-слота.

Вся информация, принимаемая модулями CPU и I/O по межмодульному интерфейсу, немедленно обрабатывается их встроенными мажорирующими узлами. Данная архитектура гарантирует корректную работу системы при отказе любого одного из модулей как на уровне CPU, так и на уровне I/O и обеспечивает ее соответствие схеме расчета надежности (см. рис. 5). В то же время в ОВК необходим механизм обмена информацией, которая имеет одиночный источник, но требует обработки в синхронном режиме аппаратуры. Такой информацией может являться диагностическое событие на любом из модулей — определение мажоритаром искажения пакета данных на одном из трех входов, срабатывание аппаратных схем контроля питания, синхронизации, внутренней диагностики. К такому же типу событий можно отнести прием ОВК пакета данных по интерфейсу Fast Ethernet. Непосредственное мажорирование для такого класса данных невозможно, однако их обработка вычислителями без дополнительной защиты может привести к отказу по общей причине. В качестве примера рассмотрим следующий сценарий. На модуле I/O происходит отказ аппаратной подсистемы. Модуль должен передать информацию об отказе для обработки процессорами, однако он передает по трем межмодульным интерфейсам различные пакеты диагностических данных. Данный сценарий нельзя считать невероятным, так как каждый отдельно взятый модуль не имеет внутренней защиты от возможных отказов его подсистем по общей причине. Такой отказ может повлиять сразу на группу выполняемых функций или несколько интерфейсов. В момент приема пакетов модулями CPU их входные мажоритары не будут иметь данных для сравнения от других модулей I/O и опасное событие

останется необнаруженным. Если принятые пакеты данных будут допущены к обработке в синхронном режиме без дополнительной проверки, это вызовет потерю логической синхронизации одновременно тремя модулями CPU и отказ всего ОВК по общей причине.

Для исключения такого сценария необходим протокол взаимодействия модулей, который гарантирует обработку процессорами только идентичных наборов данных. Пример такого взаимодействия приведен на рис. 7. Модули ОВК выполняют следующие операции:

1. На модуле I/O 1 регистрируется событие, требующее синхронной обработки.

2. Сигнал принимается блоком межмодульного обмена (MM_IO 1) и передается одновременно по трем интерфейсам модулей CPU 1, CPU 2, CPU 3.

3. Блоки межмодульного обмена MM_CPU 1, MM_CPU 2, MM_CPU 3 вычисляют хэш-функцию принятых ими пакетов данных и синхронно передают результат модулям I/O в следующем цикле обмена.

4. Каждый из блоков MM_IO 1, MM_IO 2, MM_IO 3 объединяет полученные данные в состав-

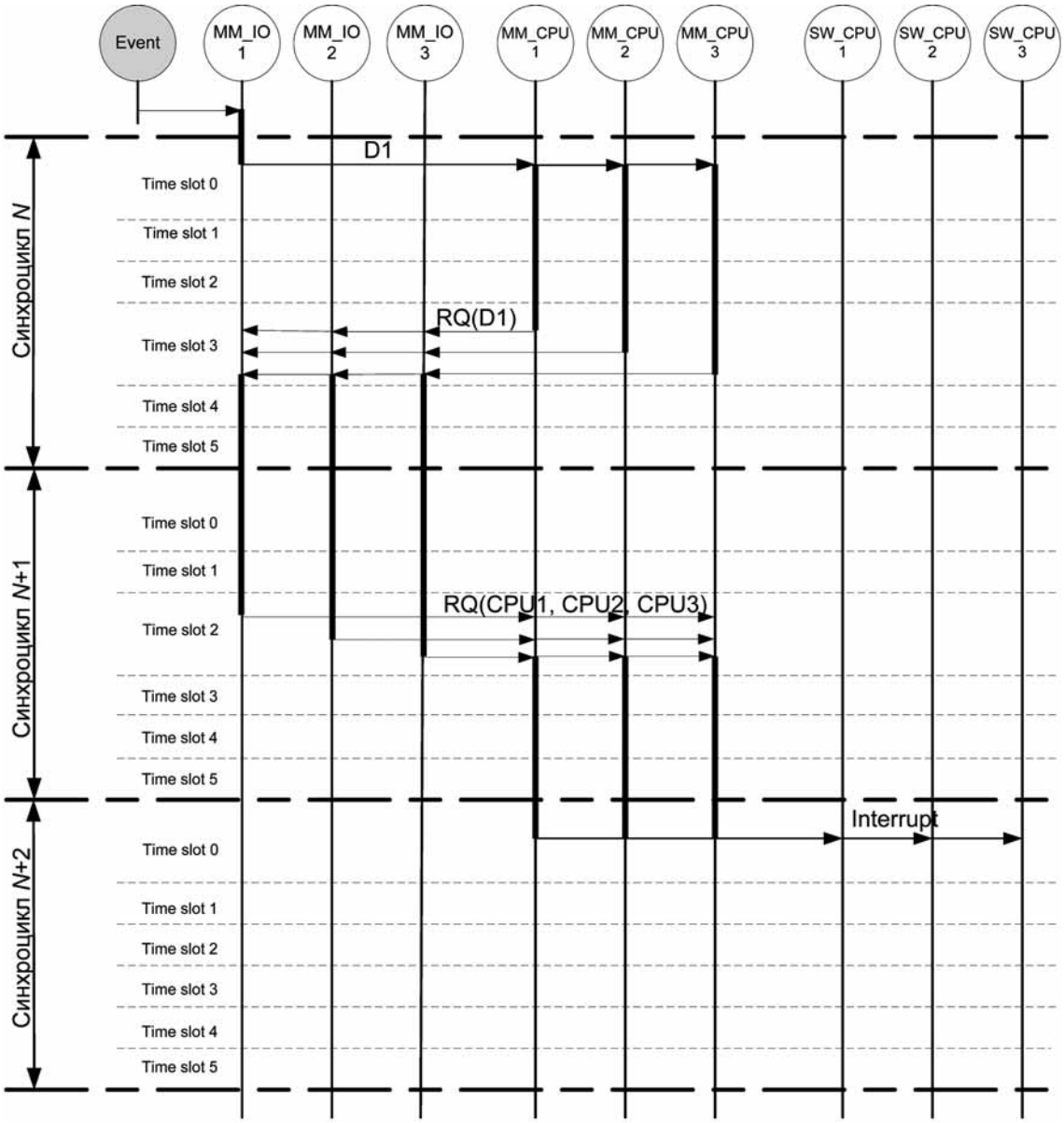


Рис. 7. UML-диаграмма обработки данных с одиночным источником по протоколу межмодульного обмена ОВК:
Event — исходное событие, зарегистрированное на одном из модулей I/O; MM_IO, MM_CPU — контроллеры межмашинного обмена модулей I/O и CPU соответственно; SW_CPU — программные обработчики прерываний модулей CPU; D1 — пакет сообщения о событии Event; RQ (D1) — группа ответных сообщений на полученный пакет D1; RQ(CPU1, CPU2, CPU3) — группа синхронных ответных сообщений, прошедших мажорирование; Interrupt — синхронное аппаратное прерывание

ной пакет и передает его в следующем цикле обмена по трем интерфейсам модулей CPU 1, CPU 2, CPU 3.

5. Блоки MM_CPU 1, MM_CPU 2, MM_CPU 3 в начале следующего вычислительного цикла синхронно генерируют прерывание своим процессорам, при этом обработчики прерывания оперируют с данными, полученными на шаге 4 и проконтролированными входными мажоритарами.

В результате выполнения этих операций каждый обработчик прерывания получает следующие входные данные:

- адрес буфера, в котором сохранен пакет данных, полученный на шаге 2 протокола обмена;
- служебный пакет, содержащий вычисленные тремя модулями CPU хэш-коды полученных на шаге 2 данных и прошедший обработку входным мажоритаром.

Данная процедура гарантирует, что полученные всеми тремя CPU данные для обработчика прерываний идентичны во всех модулях. Алгоритм обработки начинается со сравнения полученных хэш-кодов, и только в случае их совпадения исходные данные, полученные на шаге 2, передаются соответствующим программным обработчикам.

В противном случае процессоры синхронно регистрируют сбой и запускают процесс диагностики. При различии всех трех хэш-кодов содержимое буфера данных игнорируется и диагностируется сбой/отказ модуля, который являлся источником сообщения. При совпадении двух из трех хэш-кодов наиболее вероятной причиной является отказ канала межмодульного обмена либо части аппаратуры модуля CPU или I/O, непосредственно обслуживающей этот канал. В этом случае есть возможность сохранить полученные данные путем их обратной передачи по маршруту CPU $\rightarrow$ I/O $\rightarrow$ CPU с использованием мажоритаров. Поскольку процессоры в любом случае не должны оперировать с данными, которые могут различаться, эта операция должна выполняться контроллерами межмодульного обмена, которые получают от процессоров только адрес блока данных для повторной пересылки. В случае отказа канала третий процессор получит корректный блок данных по двум исправным каналам от других модулей I/O, и ОВК продолжит работу с полной конфигурацией уровня CPU. При повторной ошибке система деградирует к неполной конфигурации и продолжит работу с двумя исправными модулями.

Супервизор аварийных ситуаций

Аппаратная поддержка синхронизации модулей и распространения информации об одиночных событиях для ее синхронной обработки тремя модулями CPU обеспечивает защищенную обработку ошибок супервизором аварийных ситуаций (САС) на уровне ПО. Любое рассогласование логических состояний подсистем ОВК, обнаруженное мажоритарами или схемами аппаратного контроля, приводит к установке соответствующих флагов состояния в локальных регистрах статуса каждого из трех модулей CPU. После этого управление передается САС путем синхронной генерации немаскируемого прерывания с высшим приоритетом. Обработчики событий САС идентичны для всех модулей CPU и исполняются в синхронном режиме с мажоритарным контролем, как и любое другое ПО ОВК. Информация о состоянии резервированных подсистем ОВК представлена в каждом модуле CPU тремя комплектами регистров, по одному на подсистему. Такая структура данных позволяет САС выполнять обработку ошибок без привязки к собственному номеру в системе. К примеру, ошибка "Сбой при получении пакета данных по каналу 1 в модуле CPU 1" будет обработана модулями CPU 1, CPU 2 и CPU 3 совершенно одинаково, без каких-либо различий в кодах обработчиков. В результате обработки будет инициирована повторная передача пакета, при этом возможна выдача управляющих команд конкретной подсистеме отдельного модуля, например, "Выполнить сброс входного FIFO канала 1 модуля CPU 1". Эта команда будет передана через интерфейс межмодульного обмена с использованием схем пакетного мажорирования и принята всеми модулями, однако выполнение операции произойдет только в контроллере CPU 1, для которого она предназначена.

Независимое управление подсистемами модулей позволяет САС выполнять и более сложные операции восстановления логической синхронности, такие как выравнивание блоков данных в оперативной памяти. По команде "Синхронизовать блок памяти по адресу *A* в модуль CPU 1" контроллер межмодульного обмена CPU 1 будет переведен в режим приема данных, а контроллеры CPU 2 и CPU 3 — в режим синхронной передачи. Чтение данных в модулях CPU 2 и CPU 3 и их запись в модуль CPU 1 в режиме выравнивания проводится под управлением контроллера межмашинного обмена через собственный канал DMA, недоступный центральному процессору. На время выполнения операции тактовые частоты процессоров приостанавливаются на аппаратном уровне и возобновляются сразу после завершения копирования. С точки зрения ПО, восстановление логической синхронности модулей происходит за одну машинную команду, и единственное действие, которое нужно выполнить САС для завершения обработки, — проверить состояние регистров статуса и сформировать диагностическое сообщение об ошибке и результатах ее коррекции.

Описанные технические решения обеспечивают устойчивость ОВК к одиночным отказам модулей и исключают отказы по общей причине, однако для реализации всех допущений, принятых при расчете надежности ОВК, требуются еще два условия: постоянная фоновая диагностика модулей с эффективностью, близкой к 100 %, и автоматическое восстановление логической синхронности процессоров в случае сбоев.

Диагностика сбоев и восстановление

Мажоритары являются очень эффективным инструментом диагностики и локализации ошибок, но их использование только для контроля ввода-вывода данных ОВК не гарантирует того, что внутренние состояния процессоров взаимно идентичны. Сбои в вычислениях, не связанных с обработкой сетевых пакетов или в редко используемых областях памяти, могут долгое время оставаться незамеченными. С точки зрения общей методики оценки надежности, в этом случае мы должны учитывать интервал времени между процедурами тестирования, которые способны выявить скрытые ошибки ([1], часть 6, B.3.2.5). В наших расчетах мы принимаем этот интервал равным нулю. Для того чтобы система соответствовала данной модели, мы должны обеспечить непрерывный цикл тестирования. В ОВК для этой цели используется аппаратный механизм диагностики, работающий независимо от выполняемого ПО. Для всех данных, передаваемых по внутренней магистрали процессоров и адресов обращения к памяти, блок диагностики вычисляет хэш-функцию с 32-разрядным кодом. Полученный хэш-код передается в служебном тайм-слоте на каждом цикле обмена всем процессорам, и при обнаружении различий инициируется процедура обработки. Кроме контроля магистралей адресов и данных, хэш-функции вычисляются для всех внутренних регистров, областей памяти и других элементов, определяющих логическое состояние модуля. Полный обход всех элементов, особенно блока динамической памяти процессорных модулей, не может быть выполнен за время одного цикла синхронизации, поэтому одна итерация тестирования выполняется десятки секунд. Обмен хэш-кодами при этом происходит в каждом цикле, но данные коды обеспечивают защиту только той части данных, для которой выполнялось вычисление хэш-функции с момента начала цикла. После завершения итерации процесс тестирования начинается сначала. Интервал времени между процедурами тестирования в данном случае несоизмеримо меньше среднего времени восстановления, которое измеряется часами, поэтому при расчетах вероятности отказа этой величиной можно пренебречь.

Кроме вычисления и передачи хэш-кодов, подсистема диагностики сохраняет трассу всех выполненных за время вычислительного цикла операций с адресами всех задействованных операндов. Данная функция используется для выполнения последнего требования — восстановления логической синхронности после сбоев. При обнаружении сбоя в ОВК запускается процедура выравнивания, копирующая данные о внутреннем состоянии двух логически синхронных модулей в третий, в котором был зарегистрирован сбой. Полное копирование состояния модулей, включая динамическую память объемом 1 Гбайт, может занять несколько секунд даже при максимальном использовании пропускной способности каналов межмодульного обмена. Такая задержка может сказаться на выполнении сетевых приложений ОВК, поскольку при выравнивании выполнение функционального ПО приостанавливается. Сохранение трассы позволяет определить все участвовавшие в ошибочном цикле вычислений операнды и провести выборочное восстановление только активных областей памяти и регистров.

Таким образом, все условия, необходимые для использования при оценке вероятности отказа ОВК стандартной модели архитектуры 2oo3, были выполнены, и данная оценка применима к реализации. Соответствие аппаратуры ОВК уровню полноты безопасности SIL4 позволяет его использовать в качестве элемента сетевых компьютерных систем с максимальными требованиями по функциональной безопасности.

Список литературы

1. **ГОСТ Р МЭК** 61508—2007. Функциональная безопасность систем электрических, электронных, программируемых электронных, связанных с безопасностью.

2. **Кривошеин Б. Н.** Проектирование сетевых компьютерных систем по требованиям функциональной безопасности // Компьютерные инструменты в образовании. 2011. № 4. С. 4.

3. **IEEE Standard** for Information technology 802.3ad-2000. Telecommunications and information exchange between systems. Local and metropolitan area networks. Amendment to Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications-Aggregation of Multiple Link Segments.

4. **MIL-HDBK-217F.** Military Handbook. Reliability prediction of electronic equipment. U.S. Department of Defense, 1991.

5. **ГОСТ Р** 51901.15—2005 (МЭК 61165:1995). Менеджмент риска. Применение Марковских методов.

6. **Device** reliability report. Second quarter 2011. Xilinx, 2011. URL: http://www.xilinx.com/support/documentation/user_guides/ug116.pdf.

ИНФОРМАЦИЯ

10th IEEE EAST-WEST DESIGN & TEST SYMPOSIUM (EWDTS 2012)

Харьков, Украина, 14—17 сентября 2012 г.

Цель симпозиума **IEEE East-West Design & Test Symposium (EWDTS)** — расширение международного сотрудничества и обмен опытом между ведущими учеными Западной и Восточной Европы, Северной Америки и других стран в области автоматизации проектирования, тестирования и верификации электронных компонентов и систем.

Официальный сайт URL: http://www.ewdtest.com/conf/

УДК 004.724.4 + 519.872.3

О. И. Заяц, канд. физ.-мат. наук, доц., e-mail: zayats@amd.stu.neva.ru,
В. С. Заборовский, д-р техн. наук, проф., зав. каф., e-mail: vlad@neva.ru,
В. А. Мулюха, канд. техн. наук, асс., e-mail: vladimir@mail.neva.ru,
А. С. Вербенко, аспирант,
Санкт-Петербургский государственный политехнический университет

Управление пакетными коммутациями в телематических устройствах с ограниченным буфером при использовании абсолютного приоритета и вероятностного выталкивающего механизма. Часть 2*

Вычисляются основные технические характеристики телематической системы, описанной в первой части статьи, при произвольных значениях вероятности вытеснения α. Задача решается методом производящих функций. При емкости системы, равной k пакетам, этот метод позволяет снизить порядок решаемой системы линейных алгебраических уравнений в k/2 раз с k(k + 1)/2 до (k + 1). Исследуется зависимость вероятностей потери и сетевых задержек от параметра α. Полученные результаты объясняют ряд эффектов, наблюдающихся в натурном эксперименте, в частности, линейный закон потерь в слабозагруженной сети, а также возможность "запирания" сильнозагруженной сети по отношению к неприоритетному трафику за счет соответствующего увеличения α. Приводится пример управления мультимедиа трафиком.

Ключевые слова: *приоритетная система массового обслуживания, абсолютный приоритет, вероятностный выталкивающий механизм, пакетный трафик, управление телематическими системами*

Введение

В части 1 данной статьи была представлена математическая модель телематического устройства, обрабатывающего мультимедиа трафик, поделенный на два потока. В первый, приоритетный поток, выделен пакетный трафик, сформированный из команд управления. Все остальные пакеты отнесены ко второму потоку. Оба потока считаются простейшими, интенсивность i-го из них равняется λ_i. Первый поток обладает абсолютным приоритетом в обслуживании. Кроме того, буферная память оснащена вероятностным выталкивающим механизмом: если вся она занята, то вновь поступивший приоритетный пакет с вероятностью α выталкивает из буфера какой-либо неприоритетный и занимает его место. Длительность обработки любого пакета распределена по показательному закону с параметром μ, одинаковым для всех пакетов. Одновременно в телематическом устройстве могут находиться не более k пакетов, причем один из них обрабатыва-

* Часть 1 опубликована в журнале "Программная инженерия" № 2, 2012.

ется, а остальные ждут своей очереди. Исследуется установившийся режим работы такого устройства.

Описанная модель только типом приоритета отличается от представленной в работе [1], где он был относительным, а не абсолютным. В задачах удаленного интерактивного управления робототехническими устройствами, на решение которых ориентирована настоящая статья, реализуется именно абсолютный приоритет. Более подробный обзор литературы и детальная постановка задачи содержатся в первой части статьи.

Обозначим через N_i число пакетов i-го типа, находящихся в системе, и введем для $i = \overline{0, k}; j = \overline{0, k-i}$ вероятности

$$p_{i,j} = P\{N_1 = i,\ N_2 = j\},\ q_i = P\{N_1 = i\},\quad r_i = P\{N_1 + N_2 = i\} \tag{1}$$

и их производящую функцию $G(u, v) = \sum_{i=0}^{k} \sum_{j=0}^{k-i} p_{i,j} u^i v^j$.

В первой части статьи получено выражение

$$G(u, v) = \Big\{\mu(u - v)G(0, v) + \mu u(v - 1)G(0, 0) + [\alpha\lambda_1(u - v) + \lambda_1(1 - u)v + \lambda_2(1 - v)v] \times u \sum_{i=0}^{k} p_{i,k-i} u^i v^{k-i} + \alpha\lambda_1 u^{k+1}(v - u)p_{k,0}\Big\} \times \{[\lambda_1 u(1 - u) + \lambda_2 u(1 - v) + \mu(u - 1)]v\}^{-1}. \tag{2}$$

Кроме того, для детерминированного выталкивающего механизма ($\alpha = 1$) выведен усеченный геометрический закон распределения числа приоритетных пакетов и общего числа пакетов, аналогичный классической системе массового обслуживания (СМО) $M/M/1/k$ [2]:

$$q_i = \frac{(1 - \rho_1)\rho_1^i}{\left(1 - \rho_1^{k+1}\right)},\ r_i = \frac{(1 - \rho_1)\rho^i}{\left(1 - \rho^{k+1}\right)},\ \left(i = \overline{0, k}\right), \tag{3}$$

где $\rho_i = \lambda_i/\mu$ — коэффициент использования по i-му типу требований, а $\rho = \rho_1 + \rho_2$ — суммарный коэффициент использования.

Во второй части статьи дается метод вычисления вероятностей $p_{i,j}$, основанный на (2), доказывается, что выражения (3) сохраняют силу при произвольном $\alpha \neq 1$, вычисляются основные технические характеристики телематического устройства, приводятся числовые результаты и делаются общие выводы по всей работе в целом.

Система $\vec{M}_2/M/1/k/f_2^1$

В случае $\alpha \neq 1$ уже не удается вывести простые аналитические формулы для характеристик СМО. Однако все эти характеристики можно выразить через вероятности

$$p_i = p_{k-i,i},\ \left(i = \overline{0, k}\right), \tag{4}$$

описывающие различные состояния СМО с переполненным накопителем. Сами вероятности p_i определяются решением некоторой системы линейных алгебраических уравнений.

Покажем вначале, что через p_i выражается распределение числа приоритетных запросов. Подставляя $v = 1$ в (2) и учитывая, что $G(0, 1) = q_0$, а $p_{k,0} = q_k$, получаем для производящей функции вероятностей q_i выражение

$$G_1(u) = G_1(u, 1) = \frac{q_0 - (1 - \alpha)\rho_1 \sum_{i=0}^{k-1} p_{k-i} u^{i+1} - \rho_1 q_k u^{k+1}}{1 - \rho_1 u}. \tag{5}$$

Повторяя рассуждения первой части статьи, приходим к уравнению

$$q_0 - (1 - \alpha) \sum_{i=0}^{k-1} p_{k-i}\rho_1^{-i} - q_k\rho_1^{-k} = 0. \tag{6}$$

Вычтем левую часть (6) из числителя (5), сократим на $(1 - \rho_1 u)$ и приравняем коэффициенты при одинаковых степенях u в левой и правой части равенства. В результате получаем систему уравнений

$$q_{k-j} = (1 - \alpha) \sum_{i=1}^{j} p_i\rho_1^{i-j} + q_k\rho_1^{-j},\ \left(j = \overline{1, k}\right). \tag{7}$$

При $j = 0$ уравнение (7) также выполняется, но превращается в тривиальное тождество.

Далее просуммируем, используя (7), все вероятности q_j по j от 0 до $k - 1$, заменим эту сумму на $1 - q_k$ и решим полученное линейное уравнение относительно q_k. Это дает следующее выражение для q_k:

$$q_k = \frac{(1 - \rho_1)\rho_1^k}{\left(1 - \rho_1^{k+1}\right)}\left[1 - (1 - \alpha) \sum_{i=1}^{k} p_i \frac{\left(1 - \rho_1^{k-i+1}\right)}{(1 - \rho_1)\rho_1^{k-i}}\right]. \tag{8}$$

Естественно при $\alpha = 1$ оно удовлетворяет (3). Остальные вероятности q_j выражаются по формуле (7), которая также сводится к (3) при $\alpha = 1$.

Что касается распределения общего числа запросов N, то, подставляя в (2) значение $v = u$, приводя подобные члены и повторяя рассуждения первой части статьи, приходим к прежнему распределению общего числа запросов (3), которое, таким образом, оказывается инвариантным относительно вероятности вытеснения α. Это вполне естественно, так как вытеснение не меняет заполненность накопителя, а лишь увеличивает представительство в нем приоритетных запросов.

Теперь перейдем к получению выражений через p_i произвольных вероятностей p_{ij}. Для этого перепишем (2) в виде

$$G(u, v) = \frac{R_{10}(u, v)}{R_{20}(u, v)}, \tag{9}$$

где R_{10} и R_{20} — полиномы, степень которых относительно u равняется, соответственно, $k+2$ и 2. Эти полиномы имеют следующий вид:

$$R_{10} = (u - v)G(0, v) + u(v - 1)G(0, 0) + \\ + \alpha\rho_1(v - u)u^{k+1}p_{k,0} + \\ + [\alpha\rho_1(u-v) + \rho_1(1-u)v + \rho_2(1-v)v]u\sum_{i=0}^{k} p_{i,k-i}u^i v^{k-i}, \quad (10)$$

$$R_{20} = [\rho_1 u(1 - u) + \rho_2 u(1 - v) + (u - 1)]v.$$

Знаменатель (9) имеет два корня

$$u_{1,2} = \\ = \frac{[\rho_1 + \rho_2(1 - v) + 1] \mp \sqrt{[\rho_1 + \rho_2(1 - v) + 1]^2 - 4\rho_1}}{2\rho_1}. \quad (11)$$

По формулам Виета выполняются тождества

$$u_1 + u_2 = [\rho_1 + \rho_2(1 - v) + 1]\rho_1^{-1}, \; u_1u_2 = \rho_1^{-1}. \quad (12)$$

Корни (11) являются простыми полюсами функции (9).

Производящая функция $G(u, v)$, будучи полиномом, должна быть аналитична по u во всей плоскости. По этой причине необходимо приравнять к нулю ее вычеты в обоих полюсах

$$\operatorname*{Res}_{u=u_1} G(u, v) = 0, \; \operatorname*{Res}_{u=u_2} G(u, v) = 0. \quad (13)$$

Первое уравнение (13) сводится к равенству

$$R_{10} = (u_1, v) = 0. \quad (14)$$

Вычитая из числителя (9) левую часть (14), сократив на $(u - u_1)$, находим

$$G(u, v) = \frac{R_{11}(u, v)}{R_{21}(u, v)}, \quad (15)$$

где полиномы

$$R_{11}(u, v) = \frac{R_{10}(u, v) - R_{10}(u_1, v)}{u - u_1},$$

$$R_{21}(u, v) = -\rho_1(u - u_2)$$

имеют по u степень на единицу меньшую, соответственно, R_{10} и R_{20}.

В представлении (15) сохранился только полюс $u = u_2$. Из второго условия (13) заключаем, что

$$R_{11}(u_2, v) = 0.$$

Разность $R_{11}(u, v) - R_{11}(u_2, v)$ без остатка делится на $(u - u_2)$, получаем

$$G(u, v) = R_{12}(u, v) = -\frac{R_{11}(u, v) - R_1(u_2, v)}{\rho v(u - u_2)}.$$

Таким образом, производящая функция G может быть получена из полинома (10) применением следующего оператора разделенной разности второго порядка [3]:

$$G(u, v) = -\frac{1}{\rho v(u - u_2)} \times$$

$$\times \left[\frac{R_{10}(u, v) - R_{10}(u_1, v)}{(u - u_1)} - \frac{R_{10}(u_2, v) - R_{10}(u_1, v)}{(u_2 - u_1)}\right]. \quad (16)$$

Поскольку R_{10} есть полином относительно u, то дело сводится к применению оператора вида (16) к целым степеням u. При этом удобно использовать тождество

$$\frac{1}{(u - u_2)}\left(\frac{u^{i+2} - u_1^{i+2}}{u - u_1} - \frac{u_2^{i+2} - u_1^{i+2}}{u_2 - u_1}\right) = \sum_{j=0}^{i} u^j Q_{i-j+1},$$

где по определению

$$Q_i = \frac{u_2^i - u_1^i}{u_2 - u_1}, \quad (i \geqslant 0).$$

В первоначальном выражении (10) присутствовали вероятности состояний вида $(0, j)$ и $(i, k - i)$. После вычисления выписанной разделенной разности в (16) сохранятся лишь вероятности вида (4). Если представить производящую функцию в виде следующего полинома, расположенного по возрастающим степеням u:

$$G(u, v) = \sum_{i=0}^{k} g_i(v)u^i,$$

$$g_k = p_0, \; g_{k-1} = \rho_1^{-1}p_0 + (v - \alpha)p_1;$$

$$g_i = \Big[Q_{k-i+1} - \Big(1 + \varepsilon(1 - v)\Big)Q_{k-i}\Big]p_0 + \\ + (1 - \alpha v^{-1})p_{k-i}v^{k-i} + \sum_{j=i+1}^{k-1}\Big[(1 - \alpha v^{-1})Q_{j+1-i} + \\ + \Big(\alpha - \Big(1 + \varepsilon(1 - v)\Big)Q_{j-i}\Big)\Big]p_{k-j}v^{k-j}, \quad (17)$$

$$\Big(i = \overline{0, k-2}\Big),$$

где $\varepsilon = \rho_2/\rho_1$ характеризует соотношение интенсивностей, то, используя формулы Виета (12), нетрудно показать, что

$$Q_{j-i+1} - \Big(1 + \varepsilon(1 - v)\Big)Q_{j-i} = \\ = \rho_1^{-1}(Q_{j-i} - Q_{j-i-1}).$$

Это позволяет еще более упростить выражение (17):

$$g_k = p_0, \; g_{k-1} = \rho_1^{-1}p_0 + (v - \alpha)p_1;$$

$$g_i = \rho_1^{-1}(Q_i - Q_{i-1})p_0 + \sum_{j=1}^{i-2}\Big[\Big(\rho_1^{-1} + \alpha\Big)Q_{i-j} - \\ - \Big(\rho_1^{-1}Q_{i-j-1}\Big)\Big]p_j v^j + \Big(\rho_1^{-1} + \alpha\Big)p_{i-1}v^{i-1} - \\ - \alpha\sum_{j=0}^{i-2} Q_{i-j}p_{j+1}v^j + (1 - \alpha v^{-1})p_i v^i, \; \Big(i = \overline{2, k}\Big). \quad (18)$$

Из определения G ясно, что функции $g_i(\upsilon)$ должны быть полиномами относительно υ степени не выше, чем $(k - i)$. Следовательно, задача свелась к получению полиномиального представления для коэффициентов Q_i. Для ее решения применим общий прием, изложенный в статье [1].

Представим корни u_j в комплексной форме

$$u_j = a_j e^{i\varphi_j}, \quad \left(j = \overline{1, 2}\right),$$

где a_j и φ_j обозначают, соответственно, модуль и аргумент числа. В соответствии с (11)—(12)

$$a_1 = a_2 = \rho_1^{-1/2}, \ \varphi_1 = -\varphi_2 = \varphi,$$

где φ определяется уравнением

$$\cos(\varphi) = 1/2(\rho_1 + \rho_2(1 - \upsilon) + 1)\rho_1^{-1/2} = t(\upsilon).$$

Важно заметить, что функция $t(\upsilon)$ линейна по υ.

Из определения коэффициентов Q_j следует, что они выражаются через υ следующим образом

$$Q_j = \rho_1^{-\frac{(j-1)}{2}} U_{j-1}(t(\upsilon)), \left(j = \overline{1, k+1}\right), \qquad (19)$$

где $U_n(x)$ обозначает полином Чебышева второго рода [4]

$$U_n(\cos\varphi) = \frac{\sin[(n+1)\varphi]}{\sin\varphi}.$$

Раскладывая правую часть (19) по степеням υ, получаем

$$Q_j = \sum_{i=0}^{j-1} \frac{(-1)^i \rho_2^i}{i! 2^i \rho_1^{(i+j-1)/2}} U_{j-1}^{(i)}(t(0))\upsilon^i.$$

Производные полиномов Чебышева второго рода можно выразить [4] в виде

$$U_n^{(m)}(x) = 2^m m!\, C_{n-m}^{m+1}(x).$$

Здесь $C_n^{\upsilon}(x)$ — полином Гегенбауэра порядка n с индексом υ. В результате, окончательно находим

$$Q_j = \sum_{i=0}^{j-1} C_{j-i-1}^{i+1}(t_0)\beta^i \upsilon^i, \ \beta = -\rho_2\rho_1^{-1/2},$$
$$t_0 = t(0) = 0{,}5(1 + \rho)\rho_1^{-1/2}. \qquad (20)$$

Таким образом, в правой части уравнений (18) действительно стоят полиномы относительно υ. Вместе с тем, согласно определению производящей функции G

$$q_{k-i}(\upsilon) = \sum_{j=0}^{i} p_{k-i,j}\, \upsilon^j, \left(i = \overline{0, k}\right). \qquad (21)$$

Согласно определению (4) коэффициент при υ^i в выражении (21) должен равняться p_i. Если выразить все такие коэффициенты из (18) для $1 \leqslant i \leqslant k$, то получим систему k линейных уравнений, связывающих $(k + 1)$ вероятность (4). Коэффициенты при υ^0 совпадают тождественно, и недостающее уравнение придется выводить особо.

Опуская громоздкие, но несложные выкладки, приходим к следующим k уравнениям

$$p_i = \frac{\rho_1^{-1}\left(\rho_1^{-i} - \zeta_{i+1}\right)p_0 + \sum_{j=1}^{i-1}\left[(1-\alpha)\rho_1^{j-i-1} - \xi_{i+1,j}\right]p_j}{\xi_{i+1,i} - (1-\alpha)\rho_1^{-1}},$$

$$\left(i = \overline{1, k-1}\right), \qquad (22)$$

$$p_k = \frac{\rho_2(1 - r_k) - \alpha\rho_1(r_k - p_0) + r_0 - \rho_1^{-1}\zeta_k p_0 - \sum_{j=1}^{k-1}\xi_{k,j}p_j}{(1-\alpha)},$$

где r_n обозначает вероятности (3), а числовые коэффициенты ζ_i и $\xi_{i,j}$ выражаются через полиномы Гегенбауэра по формулам:

$$\zeta_i = \sum_{j=0}^{i-1}\left[C_{i-j-1}^{j+1}(t_0) - C_{i-j-2}^{j+1}(t_0)\right]\beta^j, \left(i = \overline{1, k}\right), \qquad (23)$$

$$\xi_{i,j} = \sum_{s=j}^{i}\left\{\rho_1^{-1}\left[C_{i-s-1}^{s-j+1}(t_0) - C_{i-s-2}^{s-j+1}(t_0)\right] - \right.$$
$$\left. - \alpha\left[C_{i-s}^{s-j+1}(t_0) - C_{i-s-1}^{s-j+1}(t_0)\right]\right\}\beta^{s-j},$$
$$\left(i = \overline{2, k}\right), \left(j = \overline{0, i-1}\right).$$

Последнее уравнение системы (23) получается из очевидного равенства

$$q_k = p_0,$$

с учетом которого на основании (8) получим

$$\frac{\left(1 - \rho_1^{k+1}\right)}{(1-\rho_1)\rho_1^k}p_0 + (1-\alpha)\sum_{i=1}^{k}\frac{\left(1 - \rho_1^{k-i+1}\right)}{(1-\rho_1)\rho_1^{k-i}}p_i = 1. \qquad (24)$$

Система (22), (24) решается численно. Если p_i получены, то остальные вероятности $p_{k-i,j}$ в (21) для $j < i$ определяются как коэффициент при υ^j в выражении (18). Соответствующие выражения $p_{k-i,j}$ для $j < i$ через вероятности (4) в целях сокращения здесь не приводятся.

Алгоритм вычислений

Перепишем систему уравнений (22), (24) в виде

$$\sum_{j=0}^{k} \vartheta_{0j} p_j = 1,\ p_i = \sum_{j=0}^{i-1} \vartheta_{ij} p_j + \vartheta_{k*}\delta_{i,\,k},\ \left(i = \overline{1,k}\right), \quad (25)$$

где ϑ_{0j} обозначает коэффициент при p_j в уравнении (24), ϑ_{ij} при $i > 0$ — аналогичный коэффициент в i-м уравнении (22), ϑ_{k*} — свободный член в k-м уравнении (22), а $\delta_{i,\,k}$ — дельта-символ Кронекера.

Матрица системы (25) квазитреугольна. Это позволяет рекуррентно выразить все p_i при $i > 0$ через p_0:

$$p_i = \gamma_i p_0 + \vartheta_{k*}\delta_{i,\,k},$$

где $\gamma_0 = 1$, а остальные γ_i пересчитываются по правилу $\gamma_i = \sum_{j=0}^{i-1} \vartheta_{ij}\gamma_j$. В результате получаем из первого уравнения (25)

$$p_i = (1 - \vartheta_{0k}\vartheta_{k*})\left(\sum_{j=0}^{k} \vartheta_{0j}\gamma_j\right)^{-1} \gamma_i + \vartheta_{k*}\delta_{i,\,k}.$$

При вычислении коэффициентов ϑ_{ij}, как это следует из результатов предыдущего раздела, все полиномы Гегенбауэра $C_n^k(t)$ берутся при одном и том же значении аргумента $t = t_0$, определяемом формулой (20). Это позволяет применить рекуррентное соотношение [4]

$$(n+1)\,C_{n+1}^k(t) = \\ = 2(n+k)\,C_n^k(t) - (n+2k-1)\,C_{n-1}^k(t),$$

отправляясь от известных значений $C_0^k(t) = 1$ и $C_1^k(t) = 2kt$.

Технические характеристики СМО легко выражаются через вероятности p_i, вероятности q_i, определяемые по формулам (7)—(8), и вероятности r_i, инвариантные относительно α и задаваемые формулами (3). Наиболее важна в сетевых задачах вероятность потери. Для приоритетных запросов потеря происходит безусловно при $N_1 = k$, а также в случае несрабатывания выталкивающего механизма при $1 \leqslant N_2 \leqslant k-1$, так что

$$P_{\text{пот}}^{(1)} = q_k + (1-\alpha)\sum_{i=1}^{k-1} p_i.$$

Запросы второго типа либо теряются на входе либо вытесняются с ожидания или обслуживания:

$$P_{\text{пот}}^{(2)} = r_k + \alpha\frac{\rho_1}{\rho_2}\sum_{i=1}^{k} p_i + \frac{\rho_1}{\rho_2}p_k.$$

Вероятность потери определяет пропускную способность системы. Различают абсолютную пропускную способность по i-му типу запросов $\lambda_i(1 - P_{\text{пот}}^{(i)})$, относительную пропускную способность для этого типа

$$\varsigma_i = 1 - P_{\text{пот}}^{(i)},\ \left(i = \overline{1,2}\right),$$

а также взаимную пропускную способность по i-му типу запросов, относительно j-го их типа

$$\varsigma_{ij} = \varsigma_i/\varsigma_j = \left(1 - P_{\text{пот}}^{(i)}\right)/\left(1 - P_{\text{пот}}^{(j)}\right).$$

Пропускной способностью можно управлять, изменяя параметр α.

Поскольку рассматриваемая СМО является эргодической, величина ς_{ij} допускает следующее толкование. Пропустим через систему достаточно большое число пакетов n вначале i-го, а затем j-го типа. Зарегистрируем число нормально принятых пакетов $n_*^{(i)}$ и $n_*^{(j)}$ каждого типа. Тогда в пределе при $n \to \infty$ отношение этих чисел будет стремиться к ς_{ij}.

Получим сетевые задержки. Обозначим через $\bar{s}_i$ среднее время пребывания в сети, а через $\bar{\tau}_i$ среднюю длину интервала между пакетами i-го типа. Введем нормированную задержку $\theta_i = \bar{s}_i/\bar{\tau}_i$, выразив $\bar{s}_i$ в долях среднего интервала между запросами i-го типа. По формулам Литтла для многопотоковых систем [5]

$$\theta_i = \frac{\bar{s}_i}{\bar{\tau}_i} = \frac{\bar{n}_{\text{оч}}^{(i)}}{\left(1 - \bar{P}_{\text{пот}}^{(i)}\right)} + \rho_i,\ \bar{\tau}_i = \frac{1}{\lambda_i},\ \left(i = \overline{1,2}\right). \quad (26)$$

Средняя длина общей очереди и средние длины каждой из частичных очередей определяются в виде $\bar{n}_{\text{оч}} = \sum_{i=1}^{k}(i-1)r_i$, $\bar{n}_{\text{оч}}^{(1)} = \sum_{i=1}^{k}(i-1)q_i$, $\bar{n}_{\text{оч}}^{(2)} = \bar{n}_{\text{оч}} - \bar{n}_{\text{оч}}^{(1)}$, а вероятность ожидания (что пакет окажется в буфере) дается равенствами

$$P_{\text{ож}}^{(1)} = 1 - q_0 - q_k,\ P_{\text{ож}}^{(2)} = 1 - r_0 - r_k.$$

Числовые результаты

Для удобства сравнения с данными работы [1] зададим ту же самую емкость буфера (30 пакетов), что соответствует $k = 31$. Вначале изучим зависимость $P_{\text{пот}}^{(i)}$ от параметра вытеснения α в режиме умеренной загрузки при преобладании неприоритетных запросов ($\rho_1 = 0{,}2$, $\rho_2 = 0{,}9$). Результаты для рассматриваемой

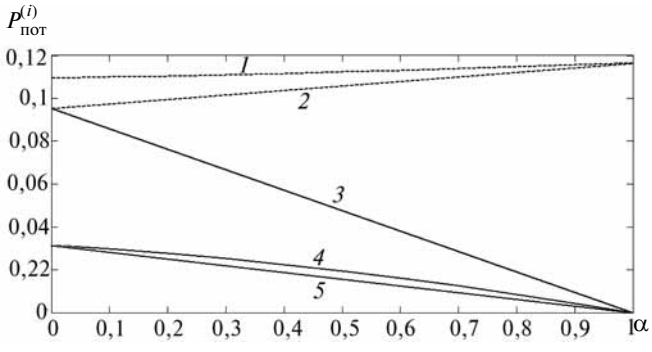


Рис. 1. Зависимость $P_{\text{пот}}^{(i)}$ от α при $\rho_1 = 0{,}2$, $\rho_2 = 0{,}9$:

1 — $i = 2$, абсолютный приоритет; *2* — $i = 2$, относительный приоритет; *3* — $i = 1$, относительный приоритет; *4* — $i = 1$, абсолютный приоритет; *5* — $i = 1$, абсолютный приоритет, линейная аппроксимация

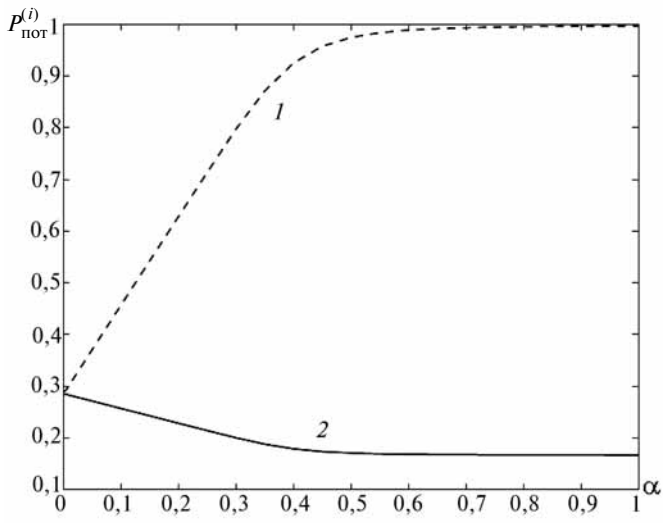


Рис. 2. Зависимость $P_{\text{пот}}^{(2)}$ от α при $\rho_1 = 1{,}2$, $\rho_2 = 0{,}2$:

1 — $i = 2$, абсолютный приоритет; *2* — $i = 1$, абсолютный приоритет

в настоящей работе системы и системы [1] представлены на рис. 1.

Граничные значения кривых *1* и *4* при $\alpha = 0$ и $\alpha = 1$, полученные численным путем, хорошо согласуются с аналитическими значениями, приведенными в статье [1]. Параметр вытеснения α позволяет эффективно управлять вероятностью потери приоритетных пакетов в очень широких пределах. При увеличении α от 0 до 1, например, значение $P_{\text{пот}}^{(1)}$ уменьшается более чем в $1{,}836 \cdot 10^{20}$ раз. При этом вероятность потери неприоритетного пакета возрастает всего лишь на 6,4 %. Следовательно, при умеренной загрузке сети и небольшой доле приоритетных пакетов вероятностный выталкивающий механизм чрезвычайно эффективно регулирует сетевые взаимодействия в пользу приоритетного трафика, не ущемляя общий трафик.

Отметим, что введение абсолютного приоритета гораздо слабее влияет на пропускную способность сети, чем изменение α. Так, при усилении приоритета от относительного до абсолютного значение $P_{\text{пот}}^{(1)}$ уменьшается всего лишь в 2—3 раза, причем примерно в одинаковой пропорции для всех α.

Важно заметить, что зависимость $P_{\text{пот}}^{(i)}(\alpha)$, полученная численным путем, очень близка к линейной для обоих типов пакетов. В практике инженерных расчетов при слабом приоритетном трафике приемлема линейная аппроксимация

$$\bar{P}_{\text{пот}}^{(i)}(\alpha) = P_{\text{пот}}^{(i)}(0) + \alpha\left(P_{\text{пот}}^{(i)}(1) - P_{\text{пот}}^{(i)}(0)\right), \qquad (27)$$

где $P_{\text{пот}}^{(i)}(0)$ и $P_{\text{пот}}^{(i)}(1)$ вычисляются аналитически. Относительная погрешность линейной аппроксимации

$$\Delta_i(\alpha) = \frac{\left|\bar{P}_{\text{пот}}^{(i)}(\alpha) - P_{\text{пот}}^{(i)}(\alpha)\right|}{P_{\text{пот}}^{(i)}(\alpha)}$$

приводится в таблице. Для приоритетного трафика она не превосходит 30 %, а для неприоритетного — 0,7 %.

Указанный линейный закон потерь наблюдается только для случая небольшого значения отношения ρ_1/ρ_2. В случае преобладания приоритетных пакетов картина кардинально меняется. На рис. 2 показана зависимость $P_{\text{пот}}^{(i)}$ от α для $\rho_1 = 1{,}2$, $\rho_2 = 0{,}2$. При преобладании приоритетного трафика зависимость $P_{\text{пот}}^{(i)}(\alpha)$ заметно отличается от линейной для обоих типов пакетов и асимптотически приближается к постоянному уровню при $\alpha \to 1$.

Отметим, что кривые на рис. 2 практически в точности повторяют аналогичные кривые работы [1], построенные для случая относительного приоритета. Следовательно, для сильно загруженной сети тип приоритета гораздо менее важен, чем наличие выталкивающего механизма и конкретное значение пара-

Относительная погрешность линейной аппроксимации

Относительная погрешность	Вероятность вытеснения α										
	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	1
$\Delta_1(\alpha)$	0	0,047	0,088	0,125	0,159	0,189	0,217	0,242	0,265	0,286	0
$\Delta_2(\alpha)$	0	0,003	0,005	0,006	0,007	0,007	0,006	0,005	0,004	0,002	0

метра вытеснения α. Выталкивающий механизм позволяет управлять сетевым трафиком и тогда, когда приоритетный механизм перестает действовать.

Разобранные выше два варианта задания сетевых параметров (ρ_1, ρ_2), взятые из работы [1], соответствуют двум принципиально разным и противоположным по своему физическому смыслу режимам передачи данных. Малые значения ρ_1 означают слабую загрузку сети приоритетными пакетами. Здесь действует простейший линейный закон потерь, аналогичный картине рис. 1. Высокий уровень ρ_1 вызывает появление перегрузок сети. В результате кривая потерь отклоняется от прямой и имеет участок насыщения в области больших α. С ростом ρ_1 нижняя граница зоны насыщения сдвигается в сторону малых α.

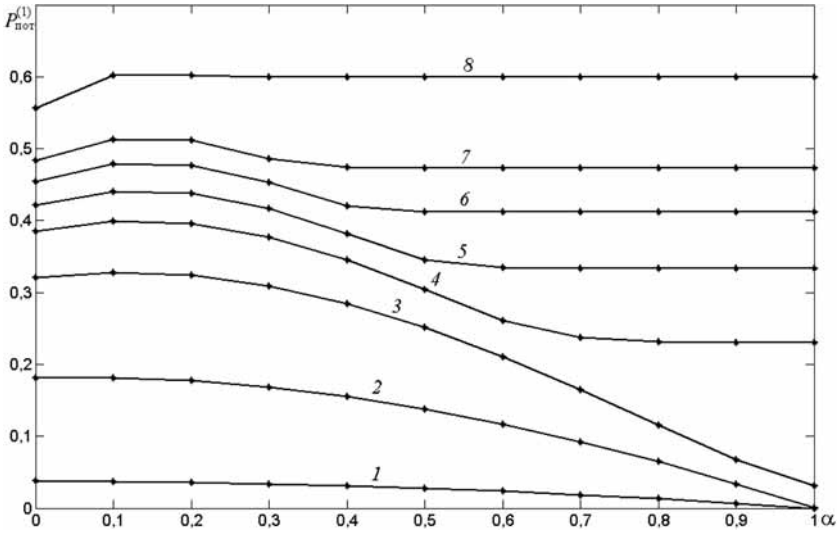


Рис. 3. Зависимость $P_{пот}^{(1)}$ от α при $\rho_2 = 1{,}5$ для различных значений ρ_1:

1 — $\rho_1 = 0{,}1$; *2* — $\rho_1 = 0{,}5$; 3 — $\rho_1 = 1{,}0$; *4* — $\rho_1 = 1{,}3$; *5* — $\rho_1 = 1{,}5$; *6* — $\rho_1 = 1{,}7$; *7* — $\rho_1 = 1{,}9$; *8* — $\rho_1 = 2{,}5$

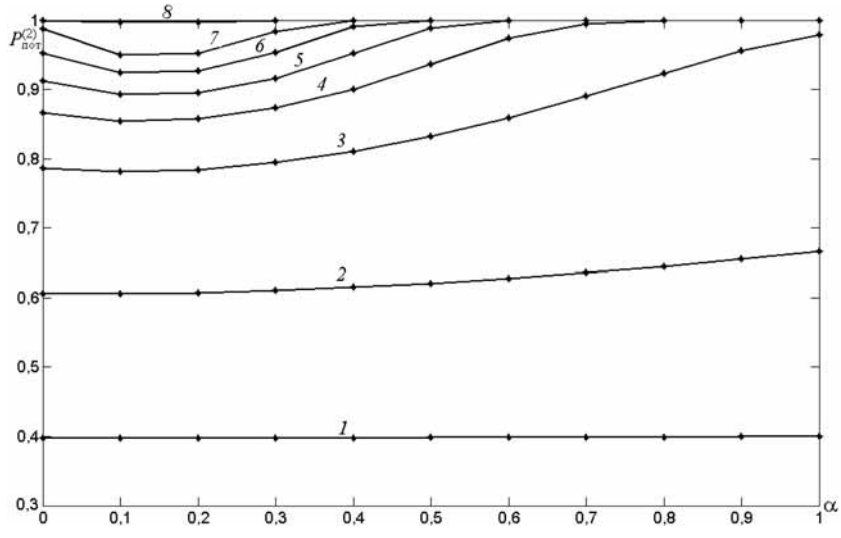


Рис. 4. Зависимость $P_{пот}^{(2)}$ от α для режимов загрузки, указанных на рис. 3

Представляет интерес изучение промежуточных режимов загрузки, в которых линейность закона потерь уже нарушена, однако зона насыщения еще не сформировалась. Для выявления таких режимов был проведен численный эксперимент, в котором значение ρ_1 варьировалось в широких пределах от 0,1 до 2,5, а значение ρ_2 поддерживалось постоянным на уровне 1,5. Целью исследования был анализ сетевых взаимодействий при увеличении приоритетного трафика на фоне постоянного и притом достаточно интенсивного потока неприоритетных пакетов.

Полученные кривые потерь для приоритетного трафика представлены на рис. 3. Задание $\rho_1 = 0{,}1$ соответствует слабо загруженной сети, для которой имеет силу линейный закон потерь. Увеличение нагрузки ρ_1 до 0,5 приводит к заметному отклонению кривой потерь от прямой, но сохраняется ее монотонность и выпуклость вверх. Пограничным является режим $\rho_1 = 1$. Монотонность функции $P_{пот}^{(1)}(\alpha)$ сохраняется, но кривая теряет выпуклость, и вблизи точки $\alpha = 1$ появляется точка перегиба. При дальнейшем увеличении ρ_1 вогнутый участок кривой в области больших α сглаживается и на его месте при $1 < \rho_1 < \rho_2$ формируется зона насыщения, где $P_{пот}^{(1)}(\alpha)$ близка к постоянной. Ордината горизонтального участка кривой увеличивается с ростом ρ_1, а в области малых α появляется локальный максимум. При дальнейшем увеличении ρ_1 область максимума сужается, а максимум разглаживается. При $\rho_1 > 2$ функция $P_{пот}^{(1)}(\alpha)$ имеет характерный для перегруженной сети вид монотонной кривой с насыщением.

Графики $P_{пот}^{(2)}(\alpha)$ для тех же режимов загрузки приведены на рис. 4. Эти графики проясняют смысл "пограничного" режима $\rho_1 = 1$. При такой загрузке в случае детерминированного вытеснения ($\alpha = 1$) имеем $P_{пот}^{(2)}(\alpha)$, близкое к единице, т. е. сеть можно практически "закрыть" для неприоритетных пакетов. При больших значениях ρ_1 эффект "закрытия" сети для обычного трафика происходит при значениях $\alpha < 1$, причем для $\rho_1 > 2$ при очень малых α. Вся сеть начинает работать только на передачу приоритетных пакетов, причем даже для них пропускная способность стабилизируется где-то на уровне всего лишь 40 % для $\rho_1 = 2{,}5$.

При $1 < \rho_1 < 2$ наблюдается неожиданный эффект в области малых значений α. Функция $P^{(1)}_{\text{пот}}(\alpha)$ имеет локальный максимум, а функция $P^{(2)}_{\text{пот}}(\alpha)$ в той же области имеет локальный минимум. В результате зависимость взаимной пропускной способности ς_{21} от α имеет вид, представленный на рис. 5. Взаимная пропускная способность обычного трафика по отношению к приоритетному имеет ярко выраженный максимум в области α порядка 10...20 %. Для $\rho_1 > 2$ значение ς_{21} пренебрежимо мало во всем диапазоне изменения α.

Полученные результаты хорошо иллюстрируют графики нормированного времени задержки приоритетных пакетов θ_1, определяемого согласно (26), представленные на рис. 6. При $\rho_1 = 0{,}1$ приоритетные пакеты практически не задерживаются в буфере, и поэтому $\theta_1 \approx \rho_1 = 0{,}1$ (второе слагаемое в (26) пренебрежимо мало). При $\rho_1 = 0{,}5$ задержка θ_1 увеличивается примерно до 1, причем слабо зависит от α. "Пограничный" режим $\rho_1 = 1$ характеризуется тем, что, во-первых, задержка начинает ощутимо зависеть от α и, во-вторых, достигает значительной величины порядка 15...20 единиц $\bar{\tau}_1$. При дальнейшем увеличении ρ_1 имеем монотонно возрастающую зависимость $\theta_1(\alpha)$ с зоной насыщения в области больших α.

Отметим, что приведенные сложные немонотонные зависимости сетевых параметров от параметра вытеснения α имеют место только при достаточно большом значении интенсивности фонового потока неприоритетных пакетов ρ_2. В работе, при тех же значениях ρ_1, был детально изучен случай $\rho_2 = 0{,}5$, отвечающий умеренной загрузке. При этом на графиках, аналогичных графикам на рис. 3—6, какие-либо экстремумы отсутствуют, а все кривые меняются монотонным образом.

Приведенные результаты показывают, что описанные сложные нелинейные эффекты сетевого взаимодействия проявляются только при достаточно большой загрузке сети, что согласуется с данными натурных наблюдений. Модели приоритетных многопотоковых СМО с вероятностным выталкивающим механизмом позволяют успешно моделировать эти эффекты и объяснять их.

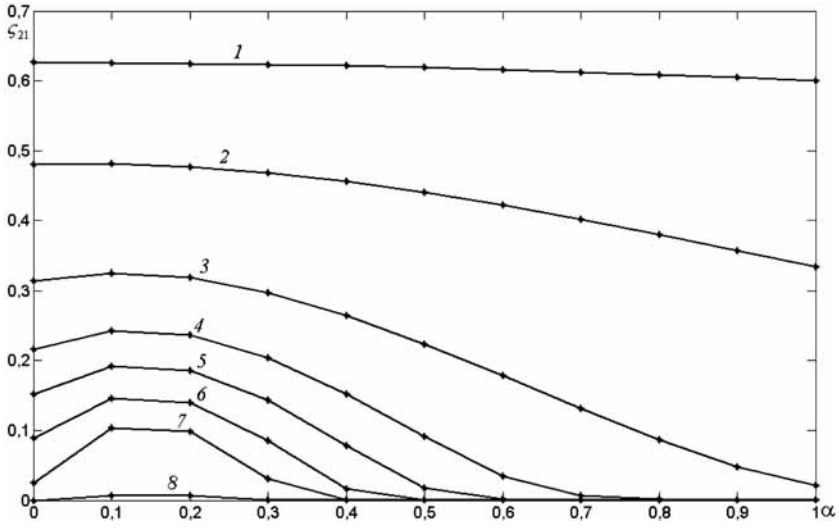


Рис. 5. Зависимость взаимной пропускной способности ς_{21} от α для режимов загрузки, указанных на рис. 3

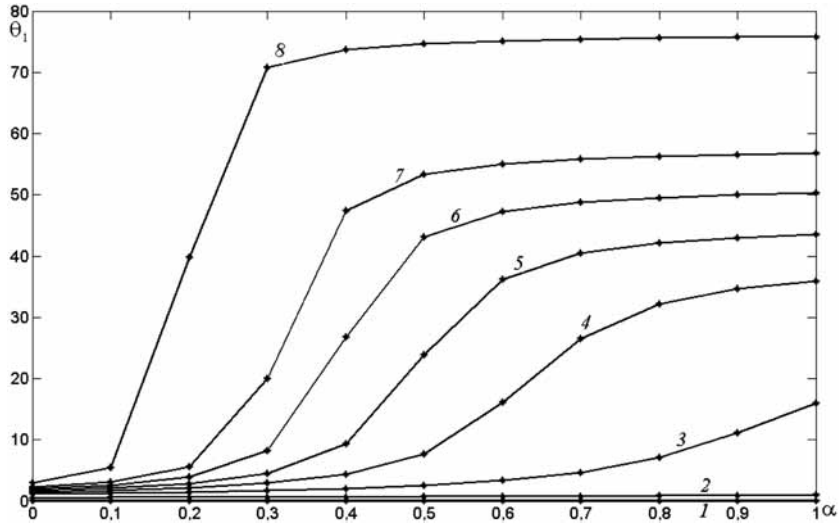


Рис. 6. Зависимость нормированного времени задержки приоритетных пакетов θ_1 от α для режимов загрузки, указанных на рис. 3

Пример управления мультимедиа потоком данных

Рассмотрим простейший пример управления двухпотоковой сетью, описанной в работе [1] и использованный при построении графиков рис. 1 ($\rho_1 = 0{,}2$, $\rho_2 = 0{,}9$). В данном случае, как мы видели, выполняется линейный закон потерь и справедлива линейная аппроксимация (27). По графикам рис. 1 получаем в случае диспетчеризации сети с абсолютным приоритетом пропускные способности

$$\varsigma_1 = 0{,}9688 + 0{,}0312\alpha,$$
$$\varsigma_2 = 0{,}8904 - 0{,}007\alpha \qquad (28)$$

и взаимную пропускную способность второго потока относительно первого

$$\varsigma_{21} = (0{,}8904 - 0{,}007\alpha)/(0{,}9688 + 0{,}0312\alpha). \qquad (29)$$

Потребуем, чтобы первый поток передавался с пропускной способностью не ниже 99 %, а пропускная способность

второго по отношению к нему была не меньше 89 %. Тогда решая неравенства

$$\varsigma_1 > 0{,}99,\ \varsigma_{21} > 0{,}89 \qquad (30)$$

относительно параметра вытеснения α, находим

$$0{,}679 < \alpha < 0{,}887. \qquad (31)$$

Если ту же самую сеть диспетчеризировать с относительным приоритетом, как это описано в работе [1], то получим вместо (28), (29) и (31), соответственно,

$$\varsigma_1 = 0{,}904 + 0{,}096\alpha,\ \varsigma_2 = 0{,}904 - 0{,}0152\alpha$$
$$\varsigma_{21} = (0{,}904 - 0{,}0152\alpha)/(0{,}904 + 0{,}096\alpha), \qquad (32)$$
$$0{,}896 < \alpha < 0{,}900,$$

т. е. в нашем случае требуется менее интенсивное вытеснение неприоритетных пакетов из накопителя, чем в модели из работы [1], что повышает гибкость управления.

В случае сильно загруженной сети вместо линейных выражений (28) и (32) приходится использовать нелинейные зависимости, подобные представленным на рисунках и получаемые численным путем. При этом приходится существенно снизить ограничения на пропускные способности по сравнению с (30) и учитывать все нюансы сложной немонотонной кривой потерь.

Заключение

В статье изложен метод расчета характеристик двухпотоковой компьютерной сети. Абсолютный приоритет по обслуживанию, а также приоритет по постановке в очередь предоставляется TCP-пакетам как управляющему виртуальному соединению, а видеопоток UDP играет роль фонового трафика. Сеть моделируется с помощью одноканальной СМО конечной емкости, снабженной вероятностным выталкивающим механизмом в комбинации с абсолютным приоритетом. Предложен эффективный вычислительный алгоритм, позволяющий сетевым инженерам и проектировщикам без больших усилий оценивать возможные варианты эксплуатации сетей.

Показано, что при умеренной загрузке, наблюдающейся, в частности, в космическом эксперименте, описанном в первой части статьи, вероятностный выталкивающий механизм является чрезвычайно эффективным средством управления высокоприоритетным трафиком, не ущемляющим при этом низкоприоритетый трафик. Чувствительность сети к выбору параметра вытеснения α на несколько порядков выше, чем к выбору типа приоритета, причем сеть можно плавно настроить на нужную пропускную способность. Имеет значение, что выталкивающий механизм управления сетью продолжает работать даже в сильно загруженной сети, когда приоритетный механизм управления резко снижает свою эффективность.

В сильно загруженных сетях пропускная способность весьма сложным образом зависит от параметра вытеснения α. При определенном соотношении параметров загрузки возможен своего рода "резонансный" режим. Взаимная пропускная способность низкоприоритетного трафика по отношению к высокоприоритетному в этом режиме имеет максимум по α.

Сеть с абсолютным приоритетом дает максимум преимуществ наиболее важным типам запросов. При этом по сравнению с относительным приоритетом можно уменьшить вероятность вытеснения α, что делает алгоритм управления более гибким.

Настоящая работа продолжает цикл статей по исследованию вероятностного выталкивающего механизма в приоритетных СМО, задуманный профессором Санкт-Петербургского государственного политехнического университета Н. О. Вильчевским и начатый работой [1]. Авторы статьи считают своим долгом вспомнить этого замечательного ученого и прекрасного человека, чутко уловившего хорошие перспективы и важность исследований на этом направлении. Данная работа посвящается светлой памяти Никиты Олеговича.

Список литературы

1. **Avrachenkov K. E., Vilchevsky N. O., Shevljakov G. L.** Priority queueing with finite buffer size and randomized push-out mechanism // Performance evaluation. 2005. Vol. 61. N 1. P. 1—16.

2. **Клейнрок Л.** Теория массового обслуживания. М.: Машиностроение, 1979.

3. **Гельфонд А. О.** Исчисление конечных разностей. М.: Физматгиз, 1959.

4. **Бейтмен Г., Эрдейн А.** Высшие трансцендентные функции. Функции Бесселя, функции параболического цилиндра, ортогональные многочлены. М.: Наука, 1974.

5. **Бронштейн О. И., Духовный И. М.** Модели приоритетного обслуживания в информационно-вычислительных системах. М.: Наука, 1976.

УДК 004.4

А. Н. Пустыгин, канд. техн. наук, доц., Челябинский государственный университет,
А. И. Иванов, д-р техн. наук, доц., нач. лаб., ФГУП "Пензенский научно-исследовательский электротехнический институт",
Ю. К. Язов, д-р техн. наук, проф., глав. науч. сотр.,
С. В. Соловьев, канд. техн. наук, доц., нач. управления, Государственный научно-исследовательский испытательный институт проблем технической защиты информации ФСТЭК, г. Воронеж,
e-mail: gniii@fstec.ru

Автоматический синтез комментариев к программным кодам: перспективы развития и применения

Рассматриваются перспективы автоматизации построения технических комментариев к программным кодам. Раскрываются пути создания генераторов технических комментариев и практические аспекты их применения.

Ключевые слова: *программный код, комментарии, автоматизация, генератор комментариев, дерево разбора*

Активное развитие цифровых информационных технологий в значительной степени обусловлено разработкой языков программирования. На настоящее время известно несколько сотен языков программирования и десятки их модификаций, и разработка новых языков программирования постоянно продолжается. При этом достаточно часто возникает необходимость трансляции программ с одного языка на другой. Может показаться, что такая трансляция должна осуществляться легко, однако это далеко не так. Программист способен переписать программу с одного языка на другой, однако для этого ему нужно понять, как работает исходная программа. Только после этого он будет способен заново написать новую программу на другом языке программирования. Попытки построить автомат-транслятор с одного языка на другой, как правило, заканчиваются неудачей. Причина в том, что в настоящее время отсутствует механизм извлечения из кода программы знаний, касающихся ее структуры, оригинальных программных решений, найденных программистом для реализации функциональных требований к программе и др. В целях облегчения понимания программы другими программистами и пользователями, для напоминания программисту-разработчику о принятых им программных решениях, исходные тексты программ сопровождаются комментариями.

Как известно, синтаксис языков программирования допускает комментарии двух видов — авторские и специализированные [1]. Последние могут генерироваться средой программирования автоматически. Как правило, авторский комментарий является текстом на естественном языке, но может содержать также элементы какой-либо формальной нотации, например математической. Важно заметить, что источником авторских комментариев является программист. При этом зачастую, по прошествии определенного времени, точное понимание принятых в программе решений и даже комментариев с трудом дается самому автору. Вместе с тем, разработка комментариев сопряжена со значительными трудностями, поскольку написание подробных комментариев означает дополнительные затраты на создание программ, а контроль полноты комментариев и их достоверности осуществлять крайне сложно.

В конечном итоге оказывается, что в больших программных проектах, созданных многочисленными коллективами программистов, комментарии к отдельным программным модулям могут быть неполными, недостоверными или просто отсутствовать. В течение жизненного цикла программного обеспечения в него вносятся изменения и дополнения. Даже если эти фрагменты имеют подробные и достоверные комментарии, трудно ожидать их согласования с комментариями, оставшимися от предшествующих версий программного обеспечения.

Следует отметить, что синтаксис комментария позволяет вводить метаописания, являющиеся эквивалентным описанием исходного кода на каком-либо метаязыке структурированного описания текста, например XML [2]. Такого рода комментарии в контексте настоящей статьи будем называть техническими. Создание программных механизмов формирования технических комментариев в настоящее время является одним из востребованных практикой направлений развития современных сред программирования.

Для генерации технических комментариев исходного текста наиболее подходящим решением представляется использование функционала компилятора с открытым исходным кодом, таким как, например, GCC [3] или javac [4]. Причина в том, что синтаксический анализатор компилятора выполняет полный разбор исходного текста в соответствии с грамматикой языка и формирует синтаксическое дерево разбора [5]. Используя синтаксическое дерево разбора, путем его десериализации можно формировать эквивалентное представление этого дерева в текстовой форме, являющееся техническим комментарием на метаязыке в заранее принятом формате. Примерная схема построения технического комментария к исходному тексту на языке программирования высокого уровня приведена на рис. 1.

Формирование технических комментариев является не только объемной, но и весьма трудоемкой процедурой, требующей автоматизации. Вместе с тем, информационный поиск показывает, что на данный момент отсутствуют средства автоматизированного синтеза комментариев к имеющемуся программному обеспечению.

Сложность создания таких средств обусловлена тем обстоятельством, что синтаксическое дерево разбора кода программы является внутренним набором данных компилятора и оно не предназначено для целей комментирования. Как следствие, необходимо сначала получить промежуточное представление исходного текста, по сути эквивалентное дереву разбора и отличающееся от него нотацией. После формирования промежуточного представления его можно обрабатывать как текст в целях получения эквивалентных представлений — технических комментариев в том или ином виде. Эквивалентные представления при этом могут быть реализованы, например, в виде блок-схемы, логической схемы алгоритма, графа потока вызовов системных сервисов, графа вложенности вызовов процедур и другими способами.

Для построения промежуточного представления могут использоваться различные известные форматы структурированного представления текста или языки разметки, такие как HTML, XML, YAML и его подмножество JSON [2].

Использование стандартных языков разметки делает возможным последующую обработку посредством существующих анализаторов текстов на языке разметки — парсеров, таких как Expat, SAX, Xerces, DOM [6].

В случае применения единого формата промежуточного представления появляется возможность построения универсального инструментария обработки такого представления исходных текстов для получения единообразных по стилю технических комментариев, мало зависящих от языка написания исходного текста. Все это, в конечном счете, позволяет создать на том или ином уровне обработки так называемые генераторы технических комментариев исходного текста программ. Технологическая схема формирования комментариев с использованием генераторов технических комментариев показана на рис. 2.

Следует подчеркнуть, что важны все три уровня комментирования, так как комментарии разного уровня могут существенно дополнять друг друга. Те решения, которые хорошо отображаются (воспринимаются) в комментариях для языка первого — высокого уровня формализации, могут оказаться намного более сложными для восприятия в комментариях для языка второго, т. е. машинного уровня, и наоборот.

Поскольку объем текстов технических комментариев может превышать размер описываемого текста, можно представлять генерируемые технические комментарии в виде самостоятельных файлов, в которых содержится информация для сопоставления исходного текста с текстом комментария.

Обычно на комментариях к программного коду программист экономит (естественное желание сократить свой интеллектуальный труд). При использовании авто-

Рис. 1. Пример схемы автоматизированного построения технического комментария

Рис. 2. Схема формирования технических комментариев с использованием генераторов комментариев

матизированных процедур комментирования затрат такого труда не требуется, необходимо только машинное время. Как следствие, занимаемый объем или степень детализации технических комментариев при таком подходе являются второстепенными характеристиками.

Обработка массивов исходных текстов с помощью генераторов технических комментариев позволит создавать библиотеки фрагментов исходных текстов (как элементарных конструкций языка программирования, так и типовых, стандартных или постоянно используемых) и комментариев к ним. С одной стороны, такой подход дает возможность построения утилит для эвристического анализа программного кода. С другой стороны, подобные библиотеки могут служить основой для систем автоматизированного (человеко-машинного) технического комментирования больших программных проектов.

Утилиты для эвристического анализа программного кода, функционирующие на основе анализа технических комментариев, дают возможность автоматического получения целого ряда сигнатур уязвимостей без анализа исходного текста, по его метаописанию, что приводит к экономии затрат труда программиста и ресурсов.

Очень упрощенным примером общепринятого на настоящее время автоматического комментирования можно считать антивирусные программы, которые хранят сигнатуры уже известных вирусов и осуществляют комментирование проверяемых фрагментов программ в терминах "ЕСТЬ" или "НЕТ" вируса. Если вирус есть, то выдается название вируса. Очевидно, что такой подход работает, однако его эффективность с течением времени монотонно снижается. Чем больше вирусов, тем больше сигнатур следует запоминать и просматривать при каждом сканировании. Обойти это неудобство удается, подписав электронной цифровой подписью (ЭЦП) уже кем-то исследованный крупный программный фрагмент (приложение). В этом случае не появляется необходимости заново исследовать приложение, достаточно проверить ЭЦП под ним. Такой подход лег в основу развития как в России, так и за рубежом одного из направлений совершенствования технологии доверенных вычислений.

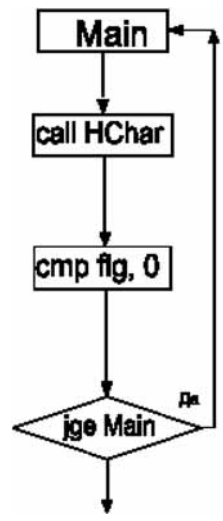
Рассмотрим пример построения технического комментария для текста на машинном языке применительно к фрагменту исходного текста на ассемблере NASM, формирующего вызов функции ввода данных из последовательного порта с проверкой результата выполнения. Исходный код фрагмента программы, текстовые комментарии к нему и содержание технического комментария представлены в таблице.

Сначала с помощью генератора эквивалентного представления, построенного на основе открытого компилятора NASM, генерируется XML-текст, описывающий исходный код на метаязыке, в котором каждый из идентификаторов становится отдельной сущностью.

Этот текст является промежуточным представлением, и для каждого идентификатора сохраняется его положение в исходном файле — номер строки, обрамленный тегами <label_1> и </label_1>. Фрагмент начинается с номера строки 183. Типы информационных тегов определяются форматом внутреннего дерева разбора компилятора. Далее с помощью известных методов анализа текста представляется возможность построить анализатор промежуточного представления (парсер XML), который разбирает это описание для извлечения информации о строении исходного текста и порождает технический комментарий.

Отличие предлагаемого подхода от уже существующих, основанных на применении статических анализаторов, состоит в построении метаязыка, пригодного, во-первых, для получения эквивалентных представлений широко круга программных систем так, чтобы анализаторы эквивалентных представлений имели по возможности наибольшей процент унифицированных модулей;

Пример построения технического комментария

Исходный текст на ассемблере	Текстовый комментарий	Промежуточное представление в виде текста XML	Технический комментарий (вариант в виде блок-схемы алгоритма)
Main:	*Метка входа*	*<label_1>183<label_2> <function_12str="17"><[Main]> </function_12>@</part>*	
call HChar	*Инструкция вызова процедуры*	*<part><label_1>187</label_1> <command_1_12str="21">call </command_1_12> <operand_12>HChar<operand_12> <label_2>numb parameter-1</label_2> </part>*	
*cmp byte [flg],*0	*Проверка возвращаемого значения*	*<part><label_1>188</label_1> <command_4_12>str<command_4_12> <operand_12>flg</operand_12> <operand_12>0</operand_12> <label_2>numb parameter-2</label_2> </part>*	
je Main	*Принятие решения о результате выполнения — нормально*	*<part><label_1>189</label_1> <command_3_12>jge<command_3_12> <operand_12>Main</operand_12> <label_2>numb parameter-1</label_2> </part>*	

во-вторых, для получения нескольких эквивалентных представлений программ для одного промежуточного их представления (например, вместо блок-схемы строить логическую схему алгоритма, граф вложенности вызовов процедур и др.).

Наличие технических комментариев позволяет на практике решать ряд других задач, кроме обеспечения понимания содержания программы, таких как:

- локализация потенциальных уязвимостей программного кода;
- обнаружение недокументированных возможностей программ;
- выявление ошибочно прокомментированных фрагментов программ;
- решение задач рефакторинга[1] программ;
- оценка качества кода программ.

Одним из принципиально важных направлений развития программирования вычислительных средств является исключение человека на уровне формализации постановки задачи. Обычное программирование предполагает, что структуру программы и связей задает человек (постановщик задачи). Как следствие, согласно традиционной технологии невозможно решать задачи, которые не до конца формализованы. При этом необходимость наличия высоко квалифицированных постановщиков задач является очень большим тормозом обычного программирования.

В некоторых областях деятельности отмеченную выше проблему удается решать за счет перехода к нейросетевым технологиям, например, путем создания программных средств для нейросетевой биометрической аутентификации личности. Для нейронных сетей уже нет необходимости в полной формализации задачи. Вместо этого достаточно дать машине примеры распознаваемых биометрических образов, используемых далее автоматом обучения, выполненным по государственному стандарту [7]. Скорость автоматического программирования (формирования текстов на машинном языке низкого уровня и исполняемых кодов при обучении нейросетевых преобразователей "биометрия—код") при этом увеличивается примерно в миллиард раз. Соответственно, увеличивается и надежность таких программ (их тестирование должно осуществляться по стандарту [8]).

С учетом изложенного выше, после введения в действие ГОСТ Р 52633.5—2011 [7] появилась возможность очень быстро создавать достаточно эффективные программные приложения. Однако, в связи с тем, что их создает автомат, они оказываются практически нечитаемыми человеком. Для приложений безопасности персональных данных это обстоятельство оказывается очень выгодным, так как выполняются требования Федерального закона № 152 "О персональных данных". Вместе с тем, для других приложений, например, комментирования нейросетевых программ, возникают трудности. Человек уже не может комментировать написанные автоматом коды без эффективного специально созданного автомата-комментатора. Комментирование нейросетевых приложений и комментирование исполняемых кодов с утраченными исходными текстами — это задачи одного уровня сложности.

По мере развития высокоразмерных нейросетевых приложений проблема их автоматического комментирования будет все более и более обостряться. Уже первые попытки создания подобных комментаторов свидетельствуют о том, что их одноуровневый вариант оказывается неработоспособен. Автоматические комментаторы нейросетевых приложений, как минимум, должны быть двухуровневыми. Они призваны параллельно создавать не только комментарии, но и нечто похожее на "исходный текст программы описания нейросетевого приложения на некотором формализованном языке программирования". Возможно, этот язык будет наследовать функции интерфейса связи внешних приложений с типовыми нейросетевыми приложениями, например, применительно к задачам обеспечения информационной безопасности в соответствии с ГОСТ Р 52633.4—2011 [9]. На текущий момент можно только констатировать острую необходимость в создании средств автоматического перевода языка машин на язык людей. Без такой "петли обратной связи" уже сейчас трудно организовать коллективную поддержку (коллективный контроль) разработки больших программных проектов, автоматизировать процессы обнаружения недокументированных возможностей программ, проводить объективную оценку качества кода программ и решать другие задачи программной инженерии.

[1] **Рефакторинг** (англ. *refactoring*) — процесс изменения внутренней структуры программы, не затрагивающий ее внешнего поведения и имеющий целью облегчить понимание ее работы. В основе рефакторинга лежит последовательность небольших эквивалентных (т. е. сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению ее согласованности и четкости.

Список литературы

1. **Энциклопедия** "SEO: Поисковая Оптимизация от А до Я" Бесплатный учебник по поисковой оптимизации и раскрутке сайтов. Комментарии (программирование). URL: http://www.seobuilding.ru/wiki/ Комментарии _(программирование) — 2010.

2. **Расширяемый** язык разметки XML. URL: http://ru.wikipedia.org/wiki/XML.

3. **Набор** компиляторов для различных языков программирования GCC. URL: http://ru.wikipedia.org/wiki/GNU_Compiler_Collection .

4. **The Java** Programming Language Compiler — javac. URL: http://download.oracle.com/javase/6/docs/technotes/guides/javac/index.html.

5. **Ахо А., Сети Р., Джеффри Д.** Компиляторы: Принципы, технологии, инструменты: пер. с англ. М.: Вильямс, 2003. 768 с.

6. **Тидвелл Д.** Основы программирования парсеров. Сообщество developerWorks. URL: http://www.ibm.com/developerworks/ru/edu/xmljava/section2.html.

7. **ГОСТ** Р 52633.5—2011 Защита информации. Техника защиты информации. Автоматическое обучение нейросетевых преобразователей биометрия — код доступа.

8. **ГОСТ** Р 52633.3—2011 Защита информации. Техника защиты информации. Тестирование стойкости средств высоконадежной биометрической защиты к атакам подбора.

9. **ГОСТ** Р 52633.4—2011 Защита информации. Техника защиты информации. Интерфейсы взаимодействия с нейросетевыми преобразователями биометрия — код.

УДК 004.4

Ф. М. Пучков, канд. физ.-мат. наук., ст. науч. сотр., e-mail: fedormex@iisi.msu.ru,
К. А. Шапченко, канд. физ.-мат. наук., ст. науч. сотр., e-mail: shapchenko@iisi.msu.ru,
Научно-исследовательский институт механики МГУ имени М. В. Ломоносова

Формальная верификация C и C++ программ: практические аспекты*

Рассматривается современное состояние исследований в области формальной верификации программного обеспечения. Предлагаются подходы по совершенствованию традиционных методов и средств верификации для автоматизированного обнаружения ошибок времени выполнения в программах на языках C и C++.

Ключевые слова: *формальная верификация программ, статический анализ, информационная безопасность*

Формальная верификация в программной инженерии

Формальная верификация (далее просто — *верификация*) представляет собой процесс определения точного соответствия между реализацией некоторой модели и ее схематическим, концептуальным описанием. Таким образом, верификация программы (или программного комплекса) позволяет удостовериться в том, что данная программа (программный комплекс) точно реализует "концептуальное" описание обслуживаемой ей (им) системы. Примерами такого "концептуального" описания являются спецификация функционирования программы на естественном или формальном языке, а также требование отсутствия в ней ошибок определенного класса.

В контексте данной статьи будем рассматривать верификацию как процесс проверки (математически строгого доказательства или опровержения) на этапе компиляции программы некоторых ее динамических свойств, в первую очередь — отсутствия **ошибок времени выполнения** (далее для краткости — ОВВ). В данной статье будем рассматривать программы, написанные на языках C и C++, поскольку они в значительной степени подвержены ОВВ. Следует отметить, что данное ограничение носит исключительно технический характер, поскольку рассматриваемые методы верификации подходят и для других императивных языков программирования (Fortran, Pascal, D, Java, C# и др.).

Методы верификации программного обеспечения используются в ряде важных областей информационных технологий, включая информационную безопасность (или безопасность информационных технологий), разработку оптимизирующих компиляторов, построение программно-аппаратных систем с повышенными требованиями к надежности. Тем не менее, широкого распространения в **программной инженерии** методы верификации пока не получили. Связано это, в первую очередь, с большой трудоемкостью процесса формальной верификации как в плане вычислительной сложности, так и ввиду сложности применяемых математических методов и алгоритмов[1]. Современные методы программной инженерии [1] мало внимания уделяют верификации программ в сравнении, например, с методами тестирования.

Дополнительным препятствием к применению методов формальной верификации при разработке программного обеспечения является отсутствие эффективных и широко распространенных средств автоматизированной верификации и обнаружения ОВВ. Традиционные средства статического анализа исходного кода недостаточно точно анализируют семантику программы и, как следствие, приводят к большому числу ложных предупреждений (ошибок первого рода)

* Работа выполнена при частичной финансовой поддержке гос. контракта 07.514.11.4116 Министерства образования и науки РФ.

[1] Следует отметить, что в общем случае задача верификации программ алгоритмически неразрешима; возможны лишь частные ее решения. Такие решения могут безошибочно работать лишь на определенных классах программ, однако, в общем случае ложные срабатывания (*ошибки первого рода*) или пропуски реальных дефектов (*ошибки второго рода*) неизбежны.

либо не позволяют гарантировать отсутствие ошибок второго рода. Зачастую число ложных срабатываний при анализе исходного кода крупных программных комплексов превышает 50 % от общего числа потенциально опасных операций, подлежащих верификации, что делает результаты анализа бесполезными для дальнейшего использования. Другой проблемой в использовании существующих средств верификации является сложность интерпретации получаемых результатов, а также отсутствие возможности проверки результатов верификации сторонними средствами, что приводит к недоверию пользователя к полученным результатам.

Для пояснения практической составляющей рассматриваемого в статье вопроса приведем несколько примеров задач, которые можно было бы решить, используя методы формальной верификации.

- Проверить (доказать либо опровергнуть), что в исследуемой программе невозможны ошибки деления на ноль. В частности, проверить, что деление на ноль невозможно в выражениях вида $expr_1/expr_2$.
- Проверить (доказать либо опровергнуть), что в исследуемой программе невозможны ошибки переполнения буфера в выражениях вида $a[i]$, где a — массив, а i — индекс.
- Проверить (доказать либо опровергнуть), что исследуемая программа гарантированно останавливается за конечное число шагов (так называемая *задача останова*).
- Проверить (доказать либо опровергнуть), что в исследуемой программе на языке C (или C++) невозможны ОВВ.

Вместе с тем, синтаксический анализ исходного кода на наличие "подозрительных" или "опасных" конструкций, вычисление количественных характеристик исходного кода (например, определение цикломатической сложности), оценка соответствия кода некоторому стилю программирования или выявление "опасной практики" программирования не являются примерами задач верификации и в данной статье не рассматриваются.

С практической точки зрения, использование средств верификации (анализаторов) в разработке программного обеспечения выглядит следующим образом. На вход такому анализатору подается исходный код программы. На выходе анализатор выдает список предупреждений — подозрительных конструкций в программе, выполнение которых может быть некорректным. Для каждой подозрительной программной конструкции анализатор может дополнительно выводить следующую информацию:

- расположение конструкции в исходном коде (имя файла, номер строки);
- тип возможной ОВВ (переполнение буфера, разыменование недействительного указателя, деление на ноль, арифметическое переполнение, неверный вызов функции и т. д.);
- значения переменных программы, состояние стека и динамической памяти (так называемый *контекст выполнения*), при которых возникает данная ОВВ;
- оценка достоверности предупреждения[2] (является эвристической характеристикой и используется, в основном, для фильтрования предупреждений с низкой оценкой достоверности).

Отметим, что анализатор может быть естественным образом совмещен с компилятором и редактором связей. Таким образом, при компиляции программы, кроме стандартных предупреждений, выдаваемых компилятором, пользователь, при желании, может просмотреть также предупреждения, выданные анализатором.

В настоящей статье предпринята попытка систематизировать существующие подходы к формальной верификации программ и предложить идеи по их практическому применению с использованием положений программной инженерии.

Современное состояние исследований по формальной верификации программ

Далее кратко описаны методы и средства формальной верификации, основанные на традиционных подходах, а также предложения авторов по их совершенствованию.

Основные теоретические подходы

Основные теоретические подходы к формальной верификации императивных процедурных программ были разработаны еще в 70—80-е гг. XX века. К ним относятся:

- проверка на модели;
- абстрактная интерпретация;
- метод индуктивных утверждений Флойда—Хоара.

Рассмотрим каждый из указанных подходов более подробно.

Проверка на модели

Проверка на модели (от англ. *model checking*) — метод автоматической формальной верификации систем переходов с конечным числом состояний [2]. Он позволяет выяснить, удовлетворяет ли заданная система переходов (модель) формальным спецификациям и построить контрпример в случае, если это не так.

В качестве модели обычно используется так называемая модель Крипке, которая формально задается следующим кортежем: $M = \langle S, S_0, R, L \rangle$, где S — множество состояний; S_0 — множество начальных состояний; $R \subseteq S$ — отношение переходов состояний; L — функция разметки, определяющая для каждого состояния системы множество пропозициональных формул, истинных в данном состоянии. Формулы задаются на множестве булевых переменных, параметризующих определенным образом состояния исследуемой системы.

[2] Оценка достоверности может быть основана, например, на оценке количества информации о контексте выполнения программы, использованной анализатором при классификации данной конструкции как возможной ОВВ.

Спецификации задаются на языке формальной временной логики, позволяющей описывать поведение системы во времени. Примерами спецификаций в линейной временной логике (от англ. *Linear Temporal Logic*, или LTL) являются формулы:

- $G(x_1 \vee x_2)$ — начиная с текущего состояния, хотя бы одна из двух переменных x_1 или x_2 всегда должна быть истинной[3];
- $(x_1 \wedge x_2)$ U x_2 — переменные x_1 и x_2 должны быть истинными, начиная с текущего состояния и до тех пор, пока не станет ложной переменная x_2[4].

Основная трудность, которую приходится преодолевать в ходе проверки на модели, связана с эффектом комбинаторного взрыва в пространстве состояний. Эта трудность возникает в системах, состоящих из многих компонентов и взаимодействующих друг с другом, а также в тех, которые обладают структурами данных, способными принимать большое число значений. Примерами таких структур данных в императивных языках программирования являются массивы и строки произвольной длины, списки, деревья и другие динамические структуры данных.

В настоящее время проверка на модели достаточно успешно применяется для верификации аппаратного обеспечения [3, 4]. В то же время верификация программного обеспечения является намного более трудоемким процессом. Это связано со следующими двумя факторами.

Во-первых, программы могут использовать динамические структуры данных, например, строки и массивы неограниченной длины, списки, деревья и другие подобные конструкции. Таким образом, число объектов, доступных программе, ограничено только размером адресного пространства программы, а число состояний этих объектов — экспонентой от размера адресного пространства (в битах), что на современных 32-битных процессорах составляет $2^{2^{35}}$. Как следствие, наиболее совершенные средства проверки на модели в настоящее время не могут оперировать моделями такого размера.

Во-вторых, семантика языков программирования (таких как C или C++) очень сложна ввиду наличия в языке рекурсивных функций, функций с произвольным количеством параметров, указателей (в том числе, арифметики указателей). Это обстоятельство приводит к тому, что построение по исходному коду модели, адекватно описывающей поведение программы, становится очень сложной задачей.

[3] Здесь G — специальный унарный оператор линейной временной логики, означающий, что следующая за ним формула истинна в текущем и всех последующих состояниях системы.

[4] Здесь U — специальный бинарный оператор линейной временной логики, означающий, что формула в левой части является истинной в текущем и всех последующих состояниях системы до тех пор, пока не станет ложной формула, стоящая в правой части выражения.

Таким образом, проверка на модели для верификации программ возможна только с использованием предварительной (ручной) оптимизации модели.

Абстрактная интерпретация

Одним из универсальных подходов к верификации программ является использование *абстрактной интерпретации*. Данный подход был разработан П. Кусо, впервые опубликован в 1977 г. в работе [5] и развит в последующих работах автора. Целью подхода на основе абстрактной интерпретации является моделирование выполнения программы, в результате которого собирается информация о ее свойствах в различных точках выполнения. А именно, в результате моделирования процесса выполнения программных инструкций значения/состояния объектов, управляемых программой (переменных, констант, динамических массивов и др.), представляются абстрактными значениями. Каждому абстрактному значению соответствует однозначно определяемое подмножество реальных значений или состояний моделируемого объекта.

Рассмотрим следующий пример. Предположим, что задана программа, манипулирующая целочисленными переменными. Пусть в каждой точке такой программы необходимо для каждой переменной предсказать, является ли ее значение положительным, отрицательным или нулевым. Рассмотрим следующую интерпретацию значений переменных, констант и арифметических операций над ними (*абстрактную семантику*). Вместо реального значения (которых бесконечно много) для каждой переменной будем хранить лишь одно из следующих семи абстрактных значений:

- V_+, соответствующее множеству положительных целых чисел;
- V_-, соответствующее множеству отрицательных целых чисел;
- V_0, соответствующее значению 0;
- V_{+-}, соответствующее множеству ненулевых целых чисел;
- V_{+0}, соответствующее множеству неотрицательных целых чисел;
- V_{-0}, соответствующее множеству неположительных целых чисел;
- V_{+-0}, соответствующее множеству всех целых чисел.

Операции над целочисленными переменными (для простоты предположим, что такими операциями являются сложение, вычитание и умножение) естественным образом доопределяются на множество пар абстрактных значений, а именно:

- $V_+ + V_+ = V_+$ (сумма двух положительных целых чисел является положительным целым числом);
- $V_+ + V_{+0} = V_+$ (сумма положительного и неотрицательного числа является положительным числом);
- $V_+ + V_- = V_{+-0}$ (сумма положительного и отрицательного чисел может быть как положительным, так и отрицательным числом, а также нулем);

и т. д.

Заканчивая рассмотрение данного примера, отметим, что в некоторых случаях использование абстрактных значений приводит к потере точности. Например, сумма двух чисел 5 и −3 является положительным числом, в то время как для соответствующих абстрактных значений знак суммы неопределен: $V_+ + V_- = V_{+-0}$. Другим недостатком метода абстрактной интерпретации является его нацеленность на получение независимых абстрактных значений для различных объектов программы, что не позволяет верифицировать зависимости, возникающие между такими объектами, например, свойства вида

$$ax + by \in V',$$

где a и b — целочисленные константы, x и y — независимые целочисленные переменные программы, а V' — некоторое абстрактное значение.

Метод индуктивных утверждений Флойда—Хоара

Метод верификации Флойда—Хоара был разработан в конце 1960-х гг. В основе этого метода лежит логика Хоара и предложенная Флойдом методика доказательства индуктивных утверждений (инвариантов цикла) [6]. В оригинале метод Флойда—Хоара предназначен для верификации соответствия (под)программы, заданной в виде исполнимой блок-схемы, ее формальной спецификации, а также для верификации гарантированного останова (под)программы. В данной статье верификация рассматривается с позиций обнаружения ОВВ в программе, поэтому изложенный далее алгоритм является модификацией классического метода Флойда—Хоара. Процесс верификации программы на предмет наличия/отсутствия в ней ОВВ состоит из следующих последовательных шагов.

Шаг 1. Рассматриваются хоаровские тройки вида $\{X\}S\{Y\}$, где X, Y — предикаты логики первого порядка, связывающие значения переменных программы и состояние ее памяти, а S — часть программы (последовательность операторов) на определенном языке. Тройка интерпретируется следующим образом. Если в некоторый момент выполнения программы был выполнен предикат X, то после выполнения последовательности операторов S должен быть выполнен предикат Y.

Шаг 2. В коде верифицируемой программы или на ее блок-схеме выбирают точки сечения таким образом, чтобы любой цикл содержал, по крайней мере, одну из них. Начальная и конечная вершины блок-схемы также объявляются точками сечения.

Шаг 3. Для каждой точки сечения i находится предикат P_i, характеризующий отношения между переменными (или другими объектами) программы в данной точке. В начале программы в качестве такого предиката выбираются предусловия анализа (под)программы, а в конце — постусловия. Такие предикаты P_i называются *индуктивными утверждениями*.

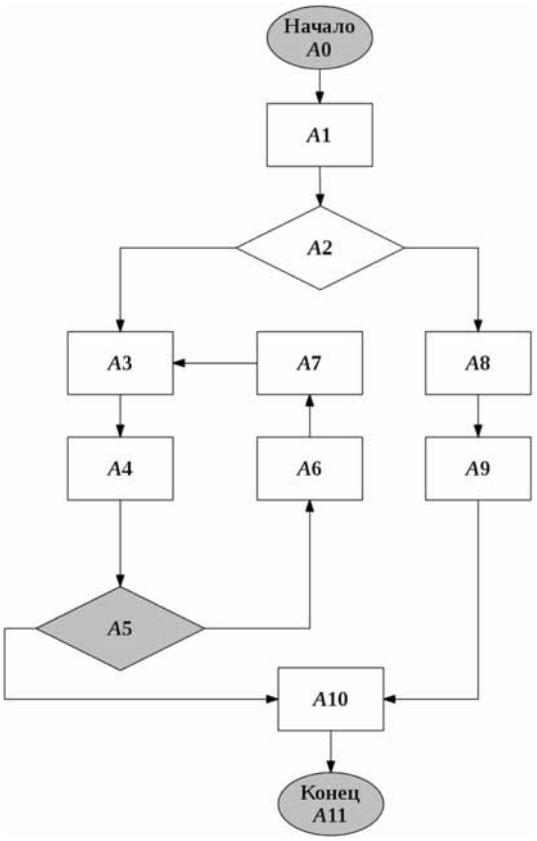


Рис. 1. Пример управляющего графа (блок-схемы) программы. Серым цветом выделены точки сечения. Граф распадается на следующие линейные участки:
$L_{0,5} = A0 \to A1 \to A2 \to A3 \to A4 \to A5$;
$L_{5,5} = A5 \to A6 \to A7 \to A3 \to A4 \to A5$;
$L_{0,11} = A0 \to A1 \to A2 \to A8 \to A9 \to A10 \to A11$;
$L_{5,11} = A5 \to A10 \to A11$

Шаг 4. В результате программа[5] разбивается на некоторое число линейных частично пересекающихся путей $L_{i,j}$, попарно соединяющих точки сечения i и j (рис. 1). Для каждого такого пути $L_{i,j}$ нужно проверить истинность хоаровской тройки $\{P_i\}L_{i,j}\{P_j\}$. Если это удается, то индуктивные утверждения P_i в точках сечения считаются доказанными.

Шаг 5. Для каждой потенциально опасной программной конструкции, требующей верификации, определяется предикат R_k, где $k \in K$ — соответствующая точка программы (блок-схемы), а K — множество всех таких точек. Предикат R_k должен содержать необходимое и достаточное условие корректности (отсутствия ОВВ) при выполнении соответствующей программной конструкции в точке k.

Шаг 6. Для каждого пути $L_{i,j}$ (где i и j — точки сечения), проходящего через потенциально опасную

[5] А точнее — множество ребер ее управляющего графа (блок-схемы).

вершину $k \in K$, формируется хоаровская тройка вида $H_{i,j,k} = \{P_i\}L'_{i,j,k}\{R_k\}$, где $L'_{i,j,k}$ — участок пути $L_{i,j}$ между вершинами i и k (рис. 1). Корректность операции в фиксированной вершине k считается доказанной, если все тройки вида $H_{i,j,k}$ при заданном фиксированном k являются истинными.

Представленный метод верификации программ на предмет наличия ОВВ является более универсальным, чем рассмотренный ранее подход на основе абстрактной интерпретации. Связано это с тем обстоятельством, что в данном случае свойства программы выражаются в виде формул логики первого порядка. Язык формул логики первого порядка является намного более выразительным, чем типовые абстрактные домены, используемые в подходе абстрактной интерпретации.

Отметим, что все шаги представленного выше алгоритма, за исключением третьего, могут быть выполнены автоматически. Однако выбор подходящего набора индуктивных утверждений требует семантического анализа программы на более высоком уровне и поэтому сложно поддается автоматизации. Вместе с тем, данное обстоятельство открывает возможности по дальнейшему совершенствованию метода верификации Флойда—Хоара, например, по использованию различных эвристик, по использованию машинного обучения, распознаванию схожих конструкций на графах и т. д.

Выводы. Каждый из трех рассмотренных выше подходов имеет свои преимущества и недостатки в зависимости от конкретной задачи. Наиболее общим из них является метод Флойда—Хоара, поскольку он позволяет доказывать и использовать более сложные свойства программы. Единственной трудностью является тот факт, что для эффективного использования метода Флойда—Хоара необходимо точно задавать индуктивные утверждения в точках сечения, что требует семантического анализа программы на более высоком уровне и трудно поддается автоматизации.

Метод абстрактной интерпретации не требует априорного задания инвариантов, однако результаты, получаемые в результате анализа программы, не могут быть улучшены путем задания пользовательских аннотаций. Кроме того, принадлежность значения переменных абстрактному значению можно, как правило, выразить простой формулой, а алгоритм поиска неподвижной точки можно использовать и в методе Флойда для "угадывания" и последующей верификации индуктивных утверждений. Обратное неверно, т. е. не любой инвариант программы можно представить в виде комбинации абстрактных значений независимых переменных программы.

Как уже отмечалось ранее, проверка на модели чаще используется для верификации аппаратного обеспечения ввиду сложности построения модели Крипке, эквивалентной верифицируемой программе на языке С или С++. Метод абстрактной интерпретации в первую очередь ориентирован на построение абстрактных значений **в отдельности** для каждого объекта, определенного в программе. Несмотря на то, что существует возможность рассматривать "многомерные" абстрактные значения, более естественным в такой ситуации представляется в качестве языка свойств программы использовать логические формулы в некоторой теории, позволяющие произвольным образом связывать элементы программы друг с другом. Таким образом, в данной статье будем в первую очередь ориентироваться на метод верификации Флойда—Хоара.

Средства автоматизированного обнаружения дефектов в программах на языках С и С++

Рассмотрим некоторые средства обнаружения дефектов в программах на языках С и С++. Поскольку в формате статьи невозможно охватить широкий перечень существующих на настоящее время подобных средств, остановимся лишь на наиболее представительных из них. Необходимо отметить, что задача сравнения различных средств статического анализа в статье не ставится. Подобное сравнение можно найти, например, в работе [7], в которой была наглядно продемонстрирована неэффективность исследованных инструментов (Flawfinder, ITS4, RATS, Splint, BOON, Archer, Polyspace C Verifier, UNO) для обнаружения уязвимостей в реальных программах. В данной работе ставится цель продемонстрировать основные достоинства и недостатки существующих анализаторов в привязке к методам верификации, лежащих в основе рассматриваемых программных комплексов с тем, чтобы в следующем разделе предложить подходы к созданию более совершенных методов и средств верификации программ на языке С.

Программный комплекс Splint

Программный комплекс Splint[6] [8, 9] (ранее — LCLint) был разработан в 2002 г. Д. Эвансом. Он представляет собой инструмент для статического анализа программ на языке С, позволяющий выявлять в них программные дефекты и уязвимости.

К видам программных дефектов, выявляемых Splint, относятся неиспользованные объявления, несоответствие типов, недостижимый код, игнорируемые возвращаемые значения, бесконечные циклы и другие типы легко выявляемых дефектов.

Более сложные проверки выполняются при наличии в исходном коде аннотаций — специальным образом оформленных комментариев. Аннотации представляют собой предположения о функциях, переменных, параметрах и типах программы.

Использование аннотаций позволяет Splint выявлять дополнительные типы программных дефектов и разновидностей опасной практики программирования, а именно:

[6] Рассматривается версия Splint 3.1.2.

- операции разыменования нулевого указателя;
- использование недействительных указателей;
- несоответствие типов в программе;
- утечки памяти и "висящие" указатели;
- опасные наложения объектов (*aliasing*);
- несоответствие интерфейсу функций;
- дефекты потока управления (бесконечные циклы, неполные switch-операторы и др.);
- переполнения буфера;
- использование опасных макроопределений;
- нарушение заданного стиля именования идентификаторов.

Не вдаваясь в детали описания представленных в перечне дефектов и разновидностей опасной практики программирования, рассмотрим применение Splint для автоматизированного обнаружения дефектов вида "переполнение буфера".

Описание метода. Для обнаружения переполнения буфера необходима как информация о размере используемого участка памяти, так и информация о смещении адреса, по которому происходит обращение к памяти, относительно начала участка памяти. Например, для проверки корректности обращения к элементу массива `a[i]` необходимо убедиться, что значение индекса `i` в данной точке программы не превышает длину массива `a`.

Программный комплекс Splint моделирует непрерывные участки памяти, используя предикаты `maxSet` и `maxRead`. Для буфера `b` (непрерывного участка памяти) выражение `maxSet(b)` обозначает наибольший адрес, считая от начала `b`, который может быть безопасно использован в операциях записи значения по этому адресу. Например, для участка памяти, соответствующего объявлению

```
char buf[MAXSIZE];
```

справедливо равенство `maxSet(buf)=MAXSIZE-1`.

Аналогично, выражение `maxRead(b)` обозначает наибольший адрес, считая от начала `b`, который может быть безопасно использован в операциях чтения значения по адресу. Например, небезопасно читать неинициализированные значения массива, а также символы строки после появления завершающего нулевого символа.

Каждый раз, когда в программе встречается операция записи значения в массив, Splint генерирует предусловие и постусловие, соответствующие данной операции. Предусловие представляет собой условие, необходимое для того, чтобы обращение к памяти считалось корректным. Постусловие представляет собой инвариант, справедливый после выполнения операции. Предположим, в программе встретилась инструкция вида:

```
buf[i] = x;
```

где `buf` — непрерывный участок памяти. Тогда Splint сгенерирует предусловие `maxSet(buf)>=i`. Для инструкции вида

```
x = buf[i];
```

средством Splint будет сгенерировано только предусловие `maxRead(buf)> = i`.

Аналогичные предусловия генерируются и для некоторых других типов синтаксических конструкций, например, для присваиваний, для объявлений массивов фиксированной длины. Для вызовов некоторых библиотечных функций Splint также использует механизм предусловий и постусловий. Предусловия гарантируют корректность работы функции, а постусловия отражают изменения, произошедшие с параметрами функции и с ее возвращаемым значением. Например, для стандартной библиотечной функции `strcpy(dest, src)` предусловие выглядит следующим образом:

```
maxSet(dest) > = maxRead(src),
```

а постусловие —

```
maxRead(dest) = = maxSet(src).
```

Алгоритм, используемый Splint, работает таким образом, что постусловия, накопленные к данной точке *A* программы, используются для проверки предусловия, сгенерированного в точке *A*. Если предусловие в точке *A* не выводится из системы накопленных постусловий, то Splint выводит сообщение о возможной уязвимости в этой точке. Для получения постусловий внутри циклов используются эвристические алгоритмы.

Примеры использования Splint. На примере табл. 1—3, рассмотрим примение Splint. Для этого будем запус-

Таблица 1

Использование Splint на примере программы `test1.c`

Файл `test1.c`	Результат работы Splint
1 #include <stdio.h> 2 #include <stdlib.h> 3 int main(void) { 4 int* buf; 5 int i, N = 10; 6 buf = malloc(N * sizeof(*buf)); 7 for(i = 0; i < N; ++i) 8 buf[i] = 0; 9 return 0; 10 }	$ splint -strict -hints test1.c Splint 3.1.2 --- 20 Feb 2009 test1.c: (in function main) **test1.c:8:5: Index of possibly null pointer buf: buf** test1.c:6:9: Storage buf may become null **test1.c:8:5: Body of for statement is not a block: buf[i] = 0;** **test1.c:9:12: Fresh storage buf not released before return** test1.c:6:3: Fresh storage buf created Finished checking --- 3 code warnings
Примечание. Строки исходной программы `test1.c` пронумерованы в целях упрощения навигации по коду. Предупреждения, выданные Splint, отмечены полужирным шрифтом.	

Таблица 2

Использование Splint на примере программы `test2.c`

Файл `test2.c`	Результат работы Splint

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void) {
4 int* buf;
5 int i, N = 10;
6 buf = malloc(N * sizeof(*buf));
7 if (buf == NULL) {
8 return 1;
9 }
10 for(i = 1; i <= N; ++i) {
11 buf[i-1] = 0;
12 }
13 free(buf);
14 return 0;
15 }
	$ splint -strict -hints test2.c
	Splint 3.1.2 --- 20 Feb 2009
	test2.c: (in function main)
	test2.c:11:5: Possible out-of-bounds store: buf[i - 1]
	Unable to resolve constraint:
	requires N @ test2.c:6:16 >= i @ test2.c:11:9
	needed to satisfy precondition:
	requires maxSet(buf @ test2.c:11:5) >= i @ test2.c:11:9 - 1
	Finished checking --- 1 code warning

Примечание. Полужирным шрифтом отмечено предупреждение (ложное), выданное Splint. Анализатор не смог доказать корректность обращения к массиву в выражении `buf[i-1]` в строке 11 тестовой программы.

Таблица 3

Использование Splint на примере программы `test3.c`

Файл `test3.c`	Результат работы Splint

1 static int f(int x) {
2 return x;
3 }
4 int main(void) {
5 int a[100], i, *p;
6 i = 100;
7 p = &i;
8 a[f(50)] = *p;
9 return 0;
10 }
	$ splint -strict -hints test3.c
	Splint 3.1.2 --- 20 Feb 2009
	test3.c: (in function main)
	test3.c:8:5: Undetected modification possible from call to unconstrained function f: f
	test3.c:8:3: Possible out-of-bounds store: a[f(50)]
	Unable to resolve constraint:
	requires f(50) @ test3.c:8:5 <= 99
	needed to satisfy precondition:
	requires maxSet(a @ test3.c:8:3) >= f(50) @ test3.c:8:5
	test3.c:8:14: Possible out-of-bounds read: *p
	Unable to resolve constraint:
	requires maxRead(&i @ test3.c:7:7) >= 0
	needed to satisfy precondition:
	requires maxRead(p @ test3.c:8:15) >= 0
	Finished checking --- 3 code warnings

Примечание. Полужирным шрифтом отмечены выданные предупреждения, из которых видно, что Splint не проводит даже простейший межпроцедурный анализ (`f(50)=50`), а также анализ указателей (`p=&i`).

кать программу `splint` с ключами `-strict` и `-hints`. Ключ `-strict` подключает все проверки, которые может делать Splint, а ключ `-hints` отключает вывод типовых подсказок, выдаваемых вместе с первым предупреждением каждого типа.

В первом примере (программа `test1.c`, табл. 1) Splint обнаружил несколько реальных дефектов программы, а именно — отсутствие проверки возвращаемого значения функции `malloc` в строке 5 и утечку памяти по указателю `buf` в строке 10. Вместе с тем, Splint доказал корректность обращения к элементу массива по индексу `i` в цикле в строке 8. Предупреждение "test1.c:8:5: Body of for statement in not a block: buf[i] = 0" является стилистическим. По этой причине его не рассматриваем.

Во втором примере (программа `test2.c`, табл. 2) исправлены дефекты, присутствующие в примере `test1.c`. Кроме того, в программу `test2.c` внесено незначительное изменение в цикл, определенный в строках 9—11. Корректность обращения к элементам массива не была нарушена. Однако в данном случае Splint уже не смог доказать корректность обращения к элементу массива в строке 10, результатом чего явилось ложное срабатывание.

В третьем примере (программа `test3.c`, табл. 3) ставилась задача проверить возможности програм-

много комплекса Splint проводить межпроцедурный анализ, а также осуществлять анализ указателей. В результате запуска Splint (три ложных срабатывания) выяснилось, что для проведения межпроцедурного анализа необходимо составление ручных аннотаций кода, а анализ указателей в этом случае неэффективен.

Представим краткие выводы по использованию программного комплекса Splint. Его достоинством является возможность верификации (хотя бы частичной) программ на языке С[7], а также высокая скорость работы. Основным недостатком Splint следует признать недостаточные глубину и точность используемого алгоритма анализа, что приводит к возникновению большого числа срабатываний при анализе реальных программ. В частности, как показал пример `test3.c`, полностью отсутствует возможность проведения межпроцедурного анализа в автоматическом режиме (т. е. без предоставления Splint аннотаций).

Авторами был также отмечен ряд недостатков технического характера программного комплекса Splint. К ним относятся: неполнота синтаксического разбора С-программ; игнорирование инициализации переменных при их объявлении; ошибки в обработке адресной арифметики (пример `test3.c`) и др.

Программный комплекс BOON

Программный комплекс BOON[8] [10, 11] предназначен для автоматического обнаружения дефектов вида "переполнение буфера" в программах на языке С. Он был разработан в 2002 г. Д. Вагнером, Д. Фостером, Э. Брюэром и А. Айкеном. В отличие от программного комплекса Splint, средство BOON представляет собой лишь исследовательский прототип, не поддерживаемый в настоящее время (по этой причине эксперименты с данным программным комплексом авторами статьи не проводились). Однако метод, используемый BOON, представляет интерес с теоретической точки зрения, поскольку является одним из первых, относящихся к числу строго формализованных.

Описание метода. Программный комплекс BOON реализует один из простейших вариантов интервального анализа — для каждой целочисленной переменной `x` вычисляется интервал `[a, b]` ее возможных значений, такой, что в каждой точке программы справедливо соотношение $a \leqslant x \leqslant b$. С этой целью:

1) для каждого указателя `p`, определенного в исходном коде, вводятся две дополнительные целочисленные переменные, содержащие информацию о размере выделенного для него участка памяти и о длине строки, содержащейся в данном участке, обозначаемые `alloc(p)` и `len(p)`, соответственно;

Таблица 4

Примеры соответствия операций на языке С интервальным уравнениям, генерируемым программным комплексом BOON

Выражение на языке С	Соответствующее интервальное уравнение
`char s[n];`	$n \subseteq alloc(s)$
`strlen(s);`	$len(s) - 1$
`strcpy(dst, src);`	$len(src) \subseteq len(dst)$
`strncpy(dst, src, n);`	$min(len(src),n) \subseteq len(dst)$
`s = "foo";`	$4 \subseteq len(s)$, $4 \subseteq alloc(s)$
`p = malloc(n);`	$n \subseteq alloc(p)$
`p = strdup(s);`	$len(s) \subseteq len(p)$, $alloc(s) \subseteq alloc(p)$
`strcat(s, suffix);`	$len(s)+len(suffix)-1 \subseteq len(s)$
`strncat(s, suffix, n);`	$len(s)+min(len(suffix)-1,n) \subseteq len(s)$
`p = getenv(...);`	$[1,\infty] \subseteq len(p)$, $[1,\infty] \subseteq alloc(p)$
`gets(s);`	$[1,\infty] \subseteq len(s)$
`fgets(s, n,...);`	$[1,n] \subseteq len(s)$
`sprintf(dst, "%s", src);`	$len(src) \subseteq len(dst)$
`sprintf(dst, "%d", n);`	$[1,20] \subseteq len(dst)$
`snprintf(dst, n, "%s", src);`	$min(len(src), n) \subseteq len(dst)$
`p[n] = '\0';`	$min(len(p), n+1) \subseteq len(p)$
`p = strchr(s, c);`	$p=s+n$; $[0, len(s)] \subseteq n$
`h = gethostbyname(...);`	$[1,\infty] \subseteq len(h->h\_name)$ $[-\infty,\infty] \subseteq h->h\_length$

2) с каждой целочисленной переменной связывается интервал, моделирующий допустимое для нее множество значений;

3) каждая операция в исходной программе преобразуется к одному или нескольким условиям, связывающим указанные интервалы операциями включения и равенства (табл. 4);

4) полученная система S интервальных уравнений решается специальным итерационным методом[9].

К недостаткам алгоритма можно отнести следующие факторы, значительно снижающие точность анализа.

- Алгоритм анализа нечувствителен к потоку управления. Анализируются лишь инструкции присваи-

[7] Верификация как математически обоснованная проверка программы на наличие ОВВ возможна при наличии в исходном коде специальных аннотаций для функций и других участков кода, а также при соответствии программы определенному набору правил.

[8] Рассматривается версия BOON 1.0.

[9] Каждая итерация состоит в том, что выбирается интервал, не удовлетворяющий системе S, который затем расширяется таким образом, что будет удовлетворять исходной системе S. Конечность такого итерационного процесса в работе [10] доказывается с использованием теоремы Тарского о неподвижной точке.

вания, причем независимо от порядка, в котором они встречаются в исследуемой программе.

- Отсутствует анализ указателей.
- Не исследуются зависимости между различными переменными.

Вместе с тем, алгоритм разрешения интервальных ограничений, используемый BOON, является корректным, что **гарантирует отсутствие пропусков реальных переполнений буферов**. Программный комплекс BOON является одним из первых средств, ориентированных на доказательство отсутствия ошибок (верификацию) в программах на языке С.

Программный комплекс Clang Analyzer

Программное средство Clang Analyzer [12] является частью инфраструктуры активно развивающегося в настоящее время компилятора Clang. Проект Clang включает в себя большую библиотеку следующих функций по статическому анализу кода:

- по работе с потоком управления (построение дерева обхода в глубину, нахождение компонентов сильной связности, альтов и других конструкций);
- по анализу зависимостей по данным;
- определения живых переменных в каждой точке;
- по анализу указателей;
- другие подобные перечисленным функции.

Следует отметить, что все эти функции нацелены в большей степени на оптимизацию программного кода и его визуализацию, чем на автоматизированное обнаружение ОВВ. Для автоматизированного обнаружения ОВВ необходимо обрабатывать более сложные логические зависимости между значениями переменных, чем просто проводить анализ зависимостей по данным или анализ указателей. На самом деле, в рамках подхода абстрактной интерпретации кроме уже перечисленных, необходимы также функции для работы с абстрактными значениями, а в рамках подхода Флойда—Хоара необходимы функции для работы с логическими формулами (инвариантами). Результаты проведенных авторами экспериментов на ряде тестовых примеров подтвердили вывод о том, что Clang Analyzer не выявляет даже простейших выходов за границу статического массива.

Программный комплекс Frama-C

Программный комплекс Frama-C [13—15] представляет собой единый интегрированный инструментарий (фреймворк) для верификации программ на языке С, объединяющий несколько методик статического анализа. Основной концепцией создания программного комплекса Frama-C является совмещение различных *корректных* методов статического анализа, что означает гарантированное отсутствие ошибок второго рода в результатах анализа. С архитектурной точки зрения, Frama-C реализован как набор плагинов, направленных на:

- исследование множества значений (вариации) для переменных программы в каждой точке выполнения (плагин `value-analysis`);
- разбиение программы на более простые участки;
- построение графа зависимостей по данным;
- решение других аналогичных перечисленным выше задач.

Плагин `value-analysis` позволяет вычислить вариацию каждой переменной в программе. Под вариацией понимается множество значений, которые переменная может принимать в данной точке программы. Плагин `value-analysis` является базовым, на нем основаны многие другие плагины.

Рассмотрим применение программного комплекса Frama-C на ряде небольших примеров с тем, чтобы исследовать его функциональные возможности. Будем использовать ключ `-val`, который указывает программе

Таблица 5

Использование Frama-C на примере программы `test3.c`

Файл `test3.c`	Результат работы Frama-C

`1 static int f(int x) {`
`2 return x;`
`3 }`
`4 int main(void) {`
`5 int a[100], i, *p;`
`6 i = 100;`
`7 p = &i;`
`8 a[f(50)] = *p;`
`9 return 0;`
`10 }`
	`$ frama-c -val test3.c`
	`[kernel] preprocessing with "gcc -C -E -I. test3.c"`
	`[value] Analyzing a complete application starting at main`
	`[value] Computing initial state`
	`[value] Initial state computed`
	`[value] Values of globals at initialization`
	`[value] computing for function f <-main.`
	`Called from test3.c:8.`
	`[value] Recording results for f`
	`[value] Done for function f`
	`[value] Recording results for main`
	`[value] done for function main`
	`[value] ====== VALUES COMPUTED ======`
	`[value] Values for function f:`
	`[value] Values for function main:`
	`a[0..49] ∈ UNINITIALIZED`
	`[50] ∈ {100; }`
	`[51..99] ∈ UNINITIALIZED`
	`i ∈ {100; }`
	`p ∈ {{ &i;}}`

Примечание. Предупреждений нет — Frama-C доказал корректность программы.	

на необходимость подключить плагин `value-analysis`.

В примере `test3.c` (табл. 5) ставилась задача проверить возможности программного комплекса Frama-C выявлять выход за границу статического массива, а кроме того — проводить межпроцедурный анализ и анализ указателей. В результате запуска Frama-C доказал корректность обеих потенциально опасных операций в программе: `a[f(50)]` и `*p`, а кроме того построил вариации для каждой из переменных.

В примере `test4.c` (табл. 6) ставилась задача проверить возможности программного комплекса Frama-C выявлять выход за границу статического массива в более сложной ситуации, когда требуется установить соотношение на переменные программы вида `i==k`. В результате запуска Frama-C не смог доказать корректность потенциально опасной операции в программе, что свидетельствует о невозможности устанавливать и эффективно использовать соотношения между различными переменными в программе. Кроме того, Frama-C имеет ограничение на количество элементов памяти, отслеживаемых при анализе, что затрудняет верификацию программ, содержащих большие массивы.

В примере `test5.c` (табл. 7) ставилась задача проверить возможности программного комплекса Frama-C

Таблица 6

Использование Frama-C на примере программы `test4.c`

Файл `test4.c`	Результат работы Frama-C

`1 int main() {`
`2 int a[10000];`
`3 int i, k, j;`
`4 for(i=0, k=0; i<100; ++i, ++k)`
`5 for(j = k; j >= 0; --j)`
`6 a[100*k+j] = k + j;`
`7 return 0;`
`8 }`
	`$ frama-c -val test4.c`
	`[kernel] preprocessing with "gcc -C -E -I. test4.c"`
	`[value] Analyzing a complete application starting at main`
	`[value] Computing initial state`
	`[value] Initial state computed`
	`[value] Values of globals at initialization`
	`test4.c:4:[value] entering loop for the first time`
	`test4.c:5:[value] entering loop for the first time`
	`test4.c:6:[value] assigning non deterministic value for the first time`
	`test4.c:6:[kernel] warning: accessing out of bounds index.`
	`assert`
	`((0 ⩽ (int)((int)(100*k)+j)) ∧`
	`((int)((int)(100*k)+j) < 10000));`
	`[value] Recording results for main`
	`[value] done for function main`
	`[dominators] computing for function main`
	`[dominators] done for function main`
	`[value] ====== VALUES COMPUTED ======`
	`[value] Values for function main:`
	`a[0..9999] ∈ [--..--] or UNINITIALIZED`
	`i ∈ {100; }`
	`k ∈ [--..--]`
	`j ∈ [-2147483648..-1] or UNINITIALIZED`

Примечание. Полужирным шрифтом выделено предупреждение (ложное) о возможном выходе за границу массива в строке 6. Для доказательства корректности операции обращения к массиву необходимо доказать, что `i==k` в строке 6 программы, что затруднительно при использовании подхода абстрактной интерпретации.

Таблица 7

Использование Frama-C на примере программы `test5.c`

Файл `test5.c`	Результат работы Frama-C

`1 #define FRAMA_C_MALLOC_INDIVIDUAL`
`2 #include "frama-c-Carbon-20110201/share/malloc.c"`
`3 int main() {`
`4 char* a;`
`5 a = malloc(100);`
`6 a[0] = 1;`
`7 a[1] = 2;`
`8 return 0;`
`9 }`
	`$ frama-c -val test5.c`
	`[kernel] preprocessing with "gcc -C -E -I. test5.c"`
	`[value] Analyzing a complete application starting at main`
	`[value] Computing initial state`
	`[value] Initial state computed`
	`[value] Values of globals at initialization`
	`[value] computing for function malloc <-main.`
	`Called from test5.c:5.`
	`[value] Recording results for malloc`
	`[value] Done for function malloc`
	`[value] Recording results for main`
	`[value] done for function main`
	`[value] ====== VALUES COMPUTED ======`
	`[value] Values for function malloc:`
	`[value] Values for function main:`
	`a ∈ {{ &Frama_C_alloc;}}`
	`Frama_C_alloc[0] ∈ {1; }`
	`[1] ∈ {2; }`
	`[2..99] ∈ {0; }`

выявлять выход за границу динамического массива. В результате запуска Frama-C доказал корректность обеих потенциально опасных операций в программе.

Таким образом, программный комплекс Frama-C представляет собой готовый прототип для верификации программ на языке С. Достоинством Frama-C является использование свободных лицензий LGPL и FreeBSD, открытость исходного кода и его доступность для исследования. Качество анализа, проводимое Frama-C, выше, чем у рассмотренных ранее средств (Splint, BOON, Clang Analyzer).

Однако в настоящее время Frama-C не подходит для анализа крупных программных продуктов. Связано это с тем ограничением, что подход, используемый Frama-C (абстрактная интерпретация), не позволяет анализировать состояние произвольного числа объектов программы. В конфигурации по умолчанию максимальное число объектов равно двухстам. В реальных программах число переменных и объем используемой памяти будут намного большими. Кроме того, анализ динамических программ неэффективен, поскольку число объектов в них заранее неизвестно.

Программный комплекс ASTRÉE

Программный комплекс ASTRÉE [15, 16] предназначен для анализа исходного кода С-программ, разработанных для встроенных систем и использующихся в составе критически важных объектов (системы управления транспортом и логистика, атомная энергетика, аэрокосмические системы и др.). Данный программный комплекс является коммерческим, поэтому эксперименты с ним авторами статьи не проводились. Вместе с тем, используемые в ASTRÉE методы представляют интерес как с теоретической, так и с практической точек зрения.

Программный комплекс ASTRÉE был создан группой разработчиков под руководством П. Кусо и основан на использовании абстрактной интерпретации. С этой целью разработана линейка абстрактных доменов (каждый следующий домен является уточнением предыдущего):

- "интервальный" абстрактный домен — значение каждой независимой переменной моделируется интервалом возможных значений;
- "восьмиугольный" абстрактный домен — значение каждой переменной, а также их всевозможные попарные разности и суммы моделируются целочисленными интервалами возможных значений;
- "многогранный" абстрактный домен — значение каждой линейной комбинации свободных переменных с целыми коэффициентами моделируется целочисленным интервалом возможных значений.

Алгоритм, используемый ASTRÉE, нацелен на обнаружение всех возможных ОВВ в программах на языке С. С учетом этого обстоятельства, он гарантирует отсутствие ошибок второго рода (пропусков реальных дефектов), однако может выдавать некоторое количество ложных срабатываний. Согласно опубликованным в статье [16] данным, среднее количество предупреждений, выдаваемых на 1000 строк исходного кода, варьируется в диапазоне от 0,6 до 1,8, что является неплохим результатом. Например, при анализе программного проекта, состоящего из 35 000 строк кода, было получено всего 22 предупреждения.

Вместе с тем, программный комплекс ASTRÉE накладывает существенные ограничения на программы, которые могут быть проанализированы с его помощью. К таким ограничениям относится невозможность анализа программ, содержащих операции динамического выделения/освобождения памяти, а также рекурсию. Такие ограничения исключают возможность верификации программ, использующих динамические структуры данных (списки, деревья, массивы и строки произвольной длины), т. е. практически все прикладные и системные программы. Исключениями являются лишь программы с фиксированной, ограниченной по размеру используемой памятью[10].

Основные работы, относящиеся к описанию и исследованию алгоритмов, используемых в ASTRÉE, содержатся в публикациях [15—18].

Подходы к созданию более совершенных средств верификации

В данном разделе рассмотрим некоторые подходы к созданию более совершенных средств верификации программ. Рассматривая причины неэффективности того или иного программного средства верификации и статического анализа, можно заметить, что многие из них реализуют некоторый фиксированный алгоритм анализа. Зная этот алгоритм, легко построить пример, для которого данный алгоритм выдаст ложный результат. Более того, реальные программные комплексы, подлежащие статическому анализу и верификации, зачастую содержат значительно более сложные конструкции, чем те, которые были использованы в качестве примеров в данной статье. По этой причине эффективная практическая верификация возможна лишь при комбинации различных подходов. Такой же вывод следует из принципиальной неразрешимости задачи полной верификации. Действительно, любое алгоритмическое решение задачи верификации заведомо будет лишь приближенным к процедуре точной верификации. Следовательно, эффективно применяемое на практике средство можно получить только путем последовательного улучшения существующих и добавления к ним новых алгоритмов статического анализа.

Другим сдерживающим фактором при использовании средств статического анализа является недоверие пользователя такого средства к результатам, которые оно выдает. Такое положение дел обусловлено как

[10] Нетрудно заметить, что программа, не содержащая рекурсию и операции динамического выделения/освобождения памяти, оперирует конечным и фиксированным количеством независимых объектов. Верификация таких программ является намного более простой задачей, чем верификация программ, в которых количество объектов заранее неизвестно.

плохой осведомленностью пользователей о принципах верификации в рамках того или иного подхода, так и отсутствием открытого исходного кода и открытой документации алгоритмов, используемых в некоторых коммерческих анализаторах.

С учетом изложенного, главной идеей, которая предлагается далее, является создание свободной программной **инфраструктуры для верификации** программного обеспечения. Следует отметить, что такой же подход реализуется в проекте Frama-C. Однако статические анализаторы Frama-C, основанные на модуле "Value Analysis", ограничены рамками подхода абстрактной интерпретации, что было продемонстрировано выше. В то же время, учитывая общность метода индуктивных утверждений Флойда—Хоара, целесообразным по мнению авторов являлось бы совмещение двух отмеченных выше подходов.

Такая единая инфраструктура верификации должна включать:

- унифицированный язык промежуточного представления программ;
- унифицированный язык представления верифицируемых свойств программы;
- обучаемый эвристический генератор пробных инвариантов (гипотез), работающий как на основе синтаксических, так и на основе семантических шаблонов;
- возможность выбора метода проверки верифицируемых свойств программы (решателя).

На рис. 2 представлена примерная архитектура программного комплекса, ориентированного на возможность расширения функциональности. Далее рассмотрим наиболее важные компоненты представленной архитектуры.

Унифицированный язык промежуточного представления. Лексический и синтаксический разбор программ на языках C и C++ является трудной задачей. Данное обстоятельство отмечается во многих исследованиях, например, в работе [19]. Связано это с большой сложностью обработки абстрактного синтаксического дерева программы, написанной на языке C или C++. В этой связи представляется целесообразным выбор единого языка промежуточного представления написанных на языках C и C++ программ и использование его в дальнейшем при разработке различных модулей представленного программного комплекса. Основными требованиями к языку промежуточного представления являются:

- выразительная эквивалентность языкам C и C++ (любой программе, написанной на языке C/C++, должна соответствовать семантически эквивалентная ей программа, записанная на языке промежуточного представления);
- возможность формализации семантики языка промежуточного представления.

В настоящее время существует несколько языков, которые можно использовать в качестве промежуточных. К их числу относятся CoreC, CIL, язык ассемблера LLVM и др. Следует отметить, что все перечисленные языки достаточно тяжело поддаются формализации. По этой причине возможна разработка нового языка, ориентированного на легкость задания его формальной семантики. Один из таких языков (IAL) был предложен в работе автора [20].

Унифицированный язык представления инвариантов. Все основные подходы к верификации (абстрактная интерпретация, метод индуктивных утверждений Флойда—Хоара) используют тот или иной язык описания свойств верифицируемой программы в различных ее точках. Рассмотренные примеры программных средств верификации также подтвержда-

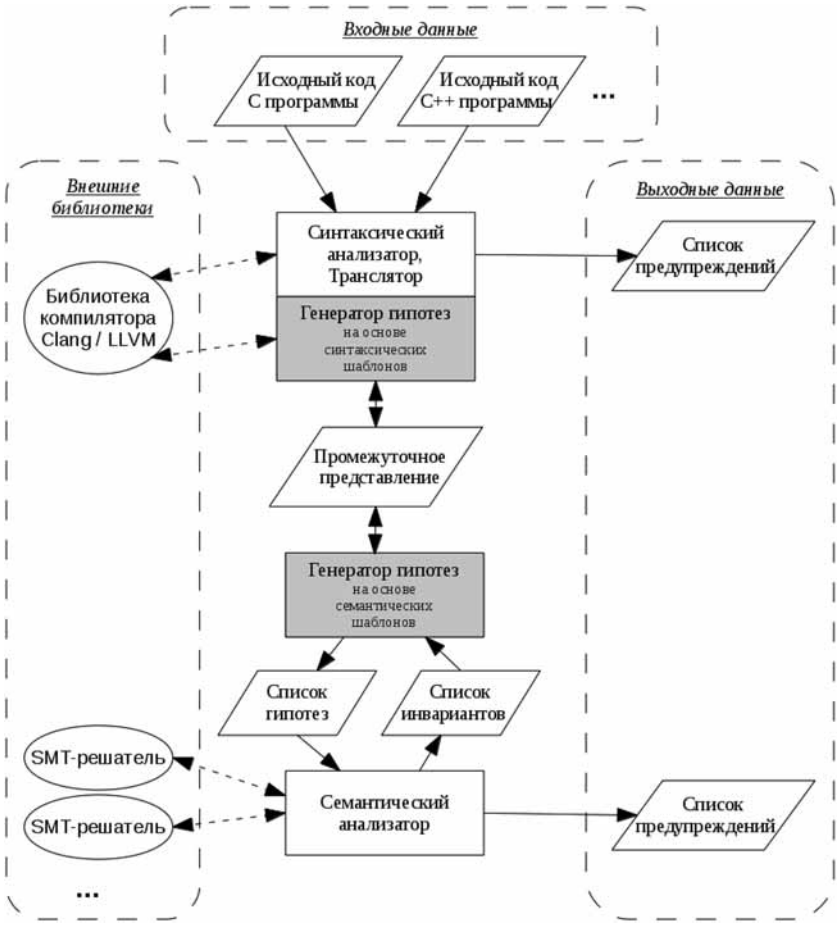


Рис. 2. Примерная архитектура программного комплекса, ориентированного на верификацию написанных на C, C++ и других программ методом Флойда—Хоара. На схеме показаны основные информационные потоки такого программного комплекса. Серым цветом отмечены компоненты — генераторы гипотез

ют этот тезис. Необходимо отметить, что простое совмещение результатов анализа (например, построение пересечения списков возможных ОВВ в программе) не является достаточно эффективным. В самом деле, если каждое средство проводит анализ программы независимо, то промежуточные факты об исследуемой программе, которые получило некоторое средство, не могут быть использованы другим средством. Если бы удалось такие промежуточные факты использовать в другом средстве, то это могло бы значительно повысить качество анализа. Такое возможно лишь в случае, когда факты, выводимые одним используемым средством понятны другому. Другими словами, для разработки расширяемой инфраструктуры верификации программ помимо унифицированного промежуточного представления необходим унифицированный язык представления инвариантов и гипотез. В его основе должна лежать некоторая логическая теория, достаточно богатая для выражения разнообразных, в том числе сложных свойств программы. Рассмотрим несколько примеров свойств программ, формализация которых необходима для эффективной верификации программ:

- размер участка памяти, выделенного под указатель `p` больше или равен `N`;
- указатель `q` ссылается на участок памяти, выделенный функцией `malloc` в некоторой строке анализируемой программы;
- указатель `node` представляет собой вершину корректно построенного списка/дерева;
- указатель `s` ссылается на корректную С-строку, длина которой не превосходит `size`.

Из представленных примеров очевидно, что указанная ранее теория должна позволять включать произвольные формулы, массивы (или неинтерпретированные функции), а также кванторы существования и единственности.

SMT-решатель. Под *SMT-формулой* (*Satisfiability Modulo Theories*) будем понимать формулу логики первого порядка, в которой:

— заданы домены для присутствующих в ней переменных;

— для некоторых функциональных и предикатных символов задана определенная интерпретация.

При этом домены переменных и интерпретация функциональных и предикатных символов должны быть согласованы с некоторой лежащей в основе теорией T. Например, в арифметике целых чисел для формулы

$$\varphi \equiv (\forall x) F(x + 1) \geqslant F(x)$$

функциональные символы "+" и "1", а также предикатный символ "$\geqslant$" обозначают соответственно операцию сложения, константу 1 и предикат "больше или равно" в их естественной интерпретации над множеством целых чисел Z. Функциональный символ F является неинтерпретируемым, т. е. допустима любая его интерпретация вида $Z \to Z$. Соответственно, формула φ утверждает, что функция F является неубывающей.

В соответствии с приведенным выше определением SMT-формулы *SMT-задачей* будем называть задачу проверки выполнимости заданной SMT-формулы, т. е. задачу определения, существуют ли такие значения свободно входящих в формулу переменных и такая интерпретация неинтерпретированных функциональных символов, что исходная формула будет истинной. Возвращаясь к рассмотренному примеру, легко установить, что формула φ является выполнимой.

SMT-решателем (или далее — прото *решателем*) для некоторой заданной теории T называется алгоритм/программа, принимающая на вход SMT-формулу этой теории, выдающая на выходе одно из трех значений:

- 1, если формула выполнима;
- 0, если формула невыполнима;
- -1, если в отведенное время решателю не удалось ни подтвердить, ни опровергнуть выполнимость формулы.

Задача верификации программы методом Флойда—Хоара (при подходящем задании индуктивных гипотез) может быть сведена к решению определенного количества SMT-задач. В данной статье не будем рассматривать подробности такого преобразования, они достаточно хорошо изложены в работах Флойда—Хоара по верификации.

Из существующих в настоящее время решателей отметим коммерческие Z3 и Yices. Указанные инструментальные средства распространяются в бинарном виде по проприетарным лицензиям. Существующие свободные аналоги (например, OpenSMT) в настоящее время не способны конкурировать с указанными двумя при проверке достаточно сложных SMT-формул. По этой причине, актуальной является разработка конкурентоспособного SMT-решателя с открытым исходным кодом, ориентированного на проверку формул целочисленной линейной арифметики с кванторами и неинтерпретированными функциональными символами (AUFLIA).

Заключение

Формальная верификация программного обеспечения является важным этапом жизненного цикла программного обеспечения. При эффективном использовании формальная верификация может значительно снизить затраты на тестирование и отладку программ. Кроме того, формальная верификация призвана упростить процедуру сертификации программы в соответствии с Руководящими документами ФСТЭК России и другими нормативными документами.

Несмотря на всю сложность использования методов формальной верификации в программной инженерии, авторы надеются, что предложенные подходы

позволят создать эффективное и удобное в использовании автоматизированное средство практической верификации программного обеспечения с открытым исходным кодом. Во многом эта надежда связана и с тем обстоятельством, что в настоящее время усилия многих разработчиков современных свободных компиляторов для программ на языках C и C++ (GCC, Clang) направлены на усложнение алгоритмов статического анализа и на повышение точности и достоверности выдаваемых по его результатам предупреждений.

Список литературы

1. **Липаев В. В.** Программная инженерия. Методологические основы. М.: Теис, 2006. 608 с.

2. **Clarke E. M., Emerson E. A., Sistla A. P.** Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Transactions on Programming Languages and Systems (TOPLAS). April 1986. Vol. 8. Issue 2. P. 244—263.

3. **Biere A., Clarke E., Raimi R., Zhu Y.** Verifying Safety Properties of a PowerPC — Microprocessor Using Symbolic Model Checking without BDDs // In Proc. 11th Int. Conf. on Computer Aided Verification, 1999. London: Springer-Verlag, 1999. P. 60—71.

4. **Dehabre D., Shankar S., Clarke E.** Model Checking VHDL with CV // Lecture Notes in Computer Science. 1998. Vol. 1522. P. 522.

5. **Cousot P., Cousot R.** Abstract Interpretation: A Uniffied Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints // In Proc. POPL'77. ACM Press, 1977. P. 238—252.

6. **Floyd R.W.** Assigning meanings to Programs // Proc. Symposium in Appl. Math. AMS, Providence. 1967. Vol. 19. Mathematical Aspects of Computer Science. P. 19—32.

7. **Zitser M.** Securing Software: An Evaluation of Static Source Code Analyzers // Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical [Computer] Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, 2003. August 29.

8. **Larochelle D., Evans D.** Statically Detecting Likely Buffer Overflow Vulnerabilities // Proc. of the 10th USENIX Security Symposium, Washington, D. C. August 13—17, 2001.

9. **Evans D., Larochelle D.** Improving Security Using Extensible Lightweight Static Analysis // IEEE Software. 2002. January/February. P. 42—51.

10. **Wagner D., Foster J., Brewer E., Aiken A.** A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities // In Network and Distributed System Security Symposium. 2000. P. 3—17.

11. **BOON** — Buffer Overrun detectiON. URL: http://www.cs.berkeley.edu/~daw/boon/.

12. **Clang** Static Analyzer. URL: http://clang-analyzer.llvm.org/.

13. **Frama-C.** URL: http://frama-c.com/.

14. **Frama-C** User Manual. URL: http://frama-c.com/download/user-manual-Carbon-20110201.pdf.

15. **Frama-C** Value Analysis Manual. URL: http://frama-c.com/download/value-analysis-Carbon-20110201.pdf.

16. **Nethercote N., Seward J.** Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation // In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, June 2007. Vol. 42. Issue 6. P. 89—100.

17. **Correnson L., Cuoq P., Puccetti A., Signoles J.** Frama-C User Manual. URL: http://frama-c.com/download/user-manual-Nitrogen-20111001.pdf.

18. **Correnson L., Cuoq P., Puccetti A., Signoles J.** Frama-C Value Analysis. URL: http://frama-c.com/download/value-analysis-Nitrogen-20111001.pdf.

19. **Baudin P., Cuoq P., Filliâtre J.-C., Marché E C., Monate B., Moy Y., Prevosto V.** ACSL: ANSI/ISO C Specification Language. [Электронный документ].

20. **Пучков Ф. М., Шапченко К. А.** Статический метод анализа программного обеспечения на наличие угроз переполнения буферов // Программирование. 2005. № 4. С. 19—34.

21. **Пучков Ф. М., Шапченко К. А.** К вопросу о выявлении возможных переполнений буферов посредством статического анализа исходного кода программ // Материалы конференции МаБИТ-04. М.: МЦНМО, 2005. С. 347—359.

22. **Пучков Ф. М., Шапченко К. А., Андреев О. О.** К созданию автоматизированных средств верификации программного кода // Материалы конференции МаБИТ-06. М.: МЦНМО, 2007. С. 401—439.

23. **Пучков Ф. М.** Средства аудита программного обеспечения на предмет обнаружения уязвимостей // Критически важные объекты и кибертерроризм. Часть 2. Аспекты программной реализации средстввия / О. О. Андреев и др.: под ред. В. А. Васенина. М.: МЦНМО, 2008. С. 375—455.

24. **Пучков Ф. М.** Методы и средства автоматизированного обнаружения уязвимостей в программах на языке С на основе статического анализа их исходных текстов // дис. ... канд. физ.-мат. наук. М.: 2010. 120 с.

ПРЕДСТАВЛЯЕМ КНИГУ

Липаев В.В.

Очерки истории отечественной программной инженерии 1940-е — 80-е годы. М.: СИНТЕГ. 2012. 245 с.

История программной инженерии в нашей стране начинается с появления ЭВМ и программирования в 1940-е годы. В очерках изложена история проектирования и производства отечественных ЭВМ, а также средств и систем автоматизации технологических процессов производства программных продуктов в 1960-е — 80-е годы. Особое внимание уделено истории разработки методов моделирования динамических объектов и стендов для тестирования и испытаний комплексов программ в реальном времени. Изложены методы оценивания качества программных продуктов, рисков, дефектов и ошибок при их разработке. Рассмотрены анализ сложности программных комплексов реального времени и распределение ресурсов ЭВМ для таких комплексов, характеристики и методы оценивания качества их компонентов. Один из разделов посвящен истории формирования экономики программной инженерии, созданию средств технико-экономического анализа и экономическому обоснованию планов разработки крупных программных продуктов.

Книга предназначена для специалистов по вычислительной технике и программной инженерии, программистов, студентов и аспирантов.

CONTENTS

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4

Дизайнер *Т.Н. Погорелова*. Технический редактор *Е.М. Патрушева*. Корректор *Т. В. Зверева*

Сдано в набор 07.03.2012 г. Подписано в печать 24.04.2012 г. Формат 60×88 1/8.
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
105120, г. Москва, ул. Нижняя Сыромятническая, д. 5/7, стр. 2, офис 2.