

Система поддержки процесса разработки и выпуска версий программного комплекса

А.А. Лаврешников, Р.Ю. Рогов, Л.Г. Тарасенко

Аннотация. Статья посвящена организации тестирования компиляторов в процессе их создания. Рассматривается структура комплекса программных средств тестирования, а также подходы к совместному проведению разработки и тестирования проектов. Приводятся различные критерии оценки этих подходов.

Введение

Процесс, в результате которого создается современное системное программное обеспечение, предполагает наличие комплексной системы обеспечения качества, действующей на всех этапах проектирования и кодирования программного продукта [1].

Конкуренция на рынке программного обеспечения заставляет искать методы, позволяющие, с одной стороны, сократить сроки разработки новых продуктов, с другой - повысить их надежность. Высокое качество и надежность поддерживаются через тестирование. Технологии быстрого тестирования обычно формируются в результате компромисса между соблюдением сроков и гарантией высокого качества [2].

В работе рассматриваются особенности организации тестирования многоязыковой мультиплатформенной системы программирования, разрабатываемой в рамках создания вычислительных средств серии «Эльбрус».

1. Общая характеристика разрабатываемого программного комплекса

Создаваемое программное обеспечение включает:

- систему компиляции с языков Фортран, С, С++, GNU C, GNU C++;
- системы статической и динамической двоичной трансляции;
- систему защищенного программирования;
- компоненты поддержки (библиотеки, линковщики, отладчики, ассемблер, дизассемблер).

Общее количество разрабатываемых компиляторов (включая технологические) в настоящий момент превышает 50, а объем исходных текстов составляет около 200 Мб. Важно отметить, что при создании значительного числа компиляторов используются общие тексты.

Ввиду того, что к работе привлечены несколько десятков программистов, одна из основных проблем состоит в организации их одновременной работы, при которой независимо внесенные изменения не приводят к общей деградации проекта.

Несмотря на то, что система программирования находится в развитии, у нее уже есть реальные пользователи, которые применяют ее компиляторные компоненты для получения кодов своих

приложений. Соответственно, развитие проекта идет с учетом текущих требований пользователей, важнейшим из которых является высокая надежность этих компонентов.

2. Организация процесса тестирования

Принцип, согласно которому организован процесс тестирования, иллюстрируется на Рис. 1. Его основными особенностями являются:

- поддержка регрессионного (не допускающего деградации) подхода к тестированию;
- повышение уровня автоматизации процессов тестирования;
- развитие комплекса средств тестирования параллельно развитию проекта.

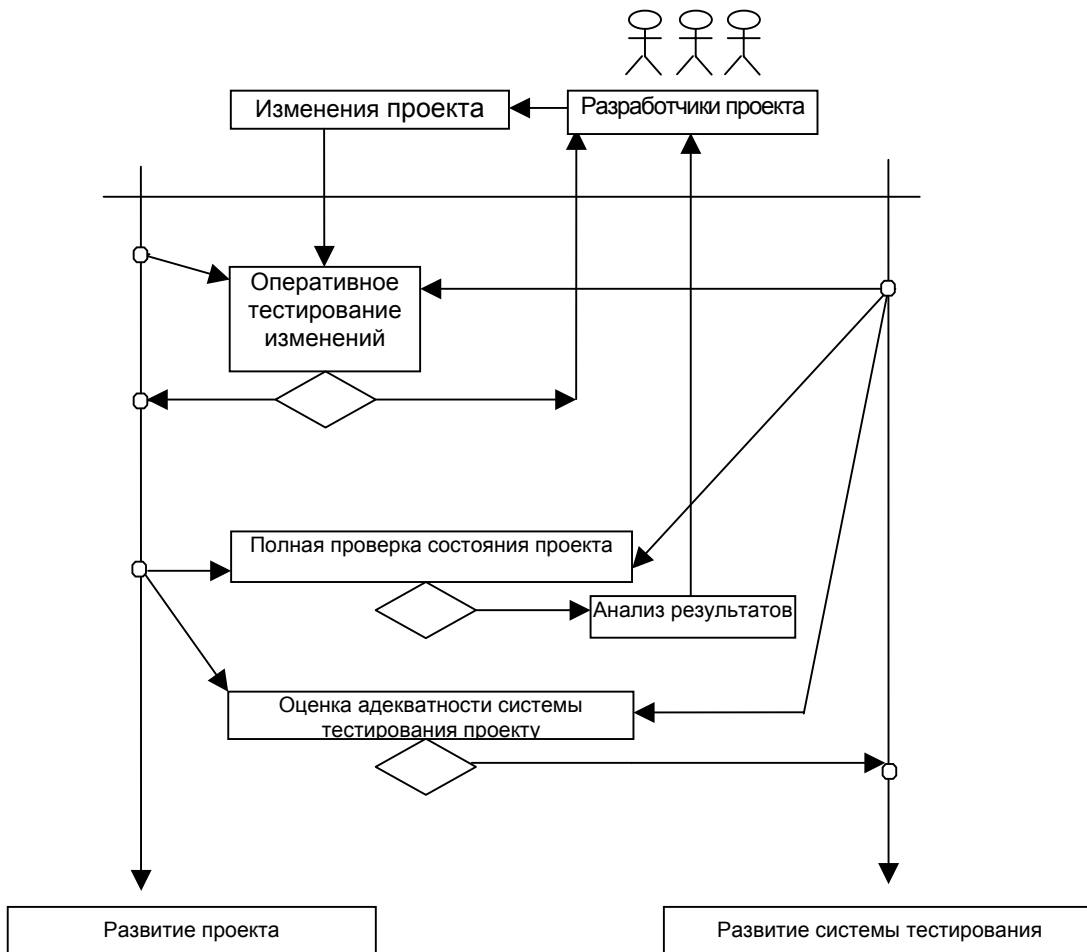


Рис. 1. Схема процесса тестирования

При внесении изменений в проект проводится автоматическое оперативное тестирование изменений. Процесс проверки изменений в случае полного тестирования всего комплекса разрабатываемых компиляторов потребовал бы больших ресурсов. Чтобы избежать этого, предусмотрена процедура, фиксирующая влияние конкретных исправлений исходных текстов на свойства того или иного компилятора. Такой анализ проводится на основе динамически формируемого графа зависимостей “исходный текст – компиляторы”. После того как определяется совокупность компиляторов, которые должны быть протестированы, осуществляется выборка из базы тестовых па-

кетов соответствующего набора. Она проводится на базе статически заданной зависимости «компилятор – тестовый набор». Эта зависимость корректируется по ходу развития проекта. Количество режимов проверки и соответственно тестовых пакетов для каждого компилятора определяется его конкретными особенностями (количеством входных языков, наличием возможности оптимизации компилируемых программ и так далее), а также - текущими потребностями процесса разработки. Естественно, для проверки всех изменений программисты не имеют возможности запускать полный тестовый пакет (время его прохождения слишком велико). Оперативное тестирование проводится с использованием пакета, обеспечивающего приемлемый на данный момент уровень надежности. При успешном результате проверки изменения вносятся в проект. В противном случае разработчики исправляют ошибки и попытка внесения изменений повторяется. Как правило, ежедневно регистрируется до нескольких десятков изменений проекта. Важными критериями, на основании которых осуществляется формирование оперативного пакета, являются время его исполнения, которое выбирается в соответствии с текущими приоритетами разработки и ресурсными возможностями, и соответствие текущих состояний тестового пакета и проекта в целом. Оценка соответствия базируется на ряде метрик, в частности на различных метриках покрытия тестами исходных текстов проекта.

Для экономии времени в системе применяется несколько уровней параллельности: уровень тестов одной совокупности, уровень тестовых прогонов различных компиляторов, уровень версий разрабатываемых компиляторов.

Периодически, как правило один раз в сутки, проводится полная проверка текущего состояния разрабатываемого проекта. Подобная практика применяется, например, в Microsoft [1].

В настоящее время поддерживается свыше 120 режимов проверки компонентов проекта. В состав полной совокупности тестовых программ входят: реальные приложения, стандартные аттестационные пакеты, тесты, получаемые в ходе отладки компиляторов – всего свыше 300 тестовых наборов. Масштабы полных планов, по которым оценивается эффективность и надежность разрабатываемых компиляторов, весьма велики – время прохождения полного тестирования занимает от нескольких часов до нескольких суток. По результатам анализа тестовых прогонов формируются конкретные предложения по улучшению надежности и эффективности разрабатываемого проекта, которые направляются разработчикам.

Кроме того, периодически проводится оценка соответствия системы тестирования текущему состоянию проекта: оценивается тестовая база, логическая структура и программная реализация системы тестирования. По результатам этого процесса выполняются необходимые модификации системы тестирования, в том числе - формирование новых тестовых программ.

3. Структура комплекса средств тестирования

Существенной особенностью проекта является тот факт, что развитие системы тестирования идет параллельно с развитием тестируемого комплекса компиляторов. Это предъявляет дополнительные требования как к структуре системы тестирования, так и к программной реализации ее отдельных компонентов. При формировании комплекса средств тестирования основными требованиями являются высокая надежность процесса тестирования и оперативность решения текущих задач по сопровождению этих средств в условиях постоянно изменяющихся требований к тестируемым программам. Важной задачей в этой связи является оптимальная технологическая организация комплекса средств тестирования, которая в данном случае отличается от традиционных систем типа [3]. Рассматриваемая система построена иерархически и имеет несколько уровней (Рис.2):

- тестовые программы;
- драйверы тестовых совокупностей;
- база сценариев тестирования;

- драйвер, формирующий задания на тестирование и осуществляющий управление драйверами тестовых совокупностей;
- база режимов тестирования;
- программный менеджер системы тестирования.

На всех уровнях используются программные средства, осуществляющие анализ информации о тестировании и формирующие отчеты различного формата.

На первом уровне системы тестирования находятся тестовые программы. Для пополнения архива тестов используется процедура контроля корректности тестовых программ. Она не только контролирует целостность многомодульных тестовых программ и проверяет корректность исходных текстов. Критерием возможности добавления теста также является его успешное применение ко всей совокупности разрабатываемых компиляторов. Это позволяет исключить ситуации блокирования внесения изменений при использовании автоматической системы.

Драйвер пакета по существу задает способ доступа к некоторой логически объединенной совокупности тестовых программ - он задает сценарий запуска каждого теста и порядок анализа его результатов. В каждом драйвере предусмотрены последовательный и параллельный режимы запуска тестов. В последовательном режиме тесты запускаются один за другим. При параллельном запуске исходный объем тестирования разбивается на определенное количество подзадач, запуск которых осуществляется одновременно. После завершения исполнения всех подзадач формируется сводный отчет о прогоне тестового пакета. Такой способ позволяет сократить время тестирования в несколько раз. Степень распараллеливания – величина переменная, она выбирается в зависимости от особенностей пакета (размер тестов, их количество), а также от текущих ресурсных возможностей. Новый драйвер включается в соответствующий архив системы тестирования, и в базе сценариев тестирования формируется ссылка соответствующего формата на этот драйвер. Под сценарием тестирования здесь понимается определенный для каждого компилятора перечень тестовых пакетов, порядок запуска тестов, входящих в эти пакеты, и порядок анализа результатов прогона.

Процесс тестирования инициируется специальным драйвером, который формирует задание на тестирование и осуществляет запуск тестов. Параметрами запуска являются тестируемый компилятор, набор его опций и сценарий тестирования. В ходе запуска формируется архив, в котором собраны результаты исполнения всех тестовых пакетов данного сценария, включая полный список

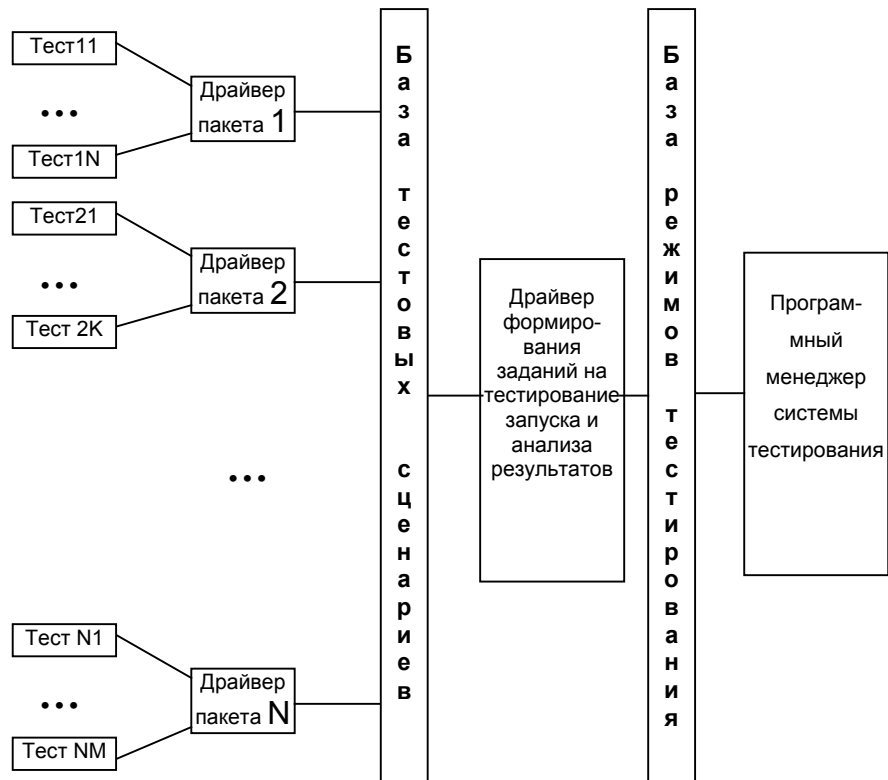


Рис.2. Структурная схема комплекса средств тестирования

запускавшихся тестов, протоколы их запуска, списки тестов, не давших положительного результата, и диагностическая информация.

Весь набор средств тестирования организован с учетом его возможных модификаций:

- включения новых тестовых программ в процесс тестирования;
- включения новых драйверов тестовых пакетов;
- включения новых объектов тестирования;
- изменения состава тестовых пакетов.

При включении в комплекс нового объекта тестирования (компилятора) в базу сценариев тестирования вводится соответствующий ему компонент с указанием объема тестирования. Ему также ставится в соответствие элемент базы режимов тестирования, задающий данные, необходимые для запуска тестового прогона, в том числе - сценарий тестирования.

В ряде случаев в ходе разработки возникает необходимость коррекции тестовых наборов для некоторых режимов тестирования. Вносятся много изменений, временно понижающих надежность компилятора в данном режиме, изменяется комплекс средств отладки и прочее. Тем не менее и в этих условиях необходимо продолжать разработку. Для таких случаев предусмотрена автоматически сопровождаемая база коррективов, работа с которой поддерживается на уровне драйверов тестовых пакетов. Это позволяет не корректировать архив тестовых программ, а просто задавать текущие списки исключений для определенного режима тестирования. Для других же режимов тестирования список тестов остается неизменным.

Общее управление процессом автоматического тестирования осуществляет программный менеджер. При тестировании изменений в исходных текстах проекта он определяет перечень компиляторов, которые необходимо проверить, и параметры тестирования, после чего инициирует прогон. По результатам проверки либо вносятся изменения в проект, либо выдается диагностическое сообщение о найденных ошибках - в этом случае все материалы прогонов сохраняются в соответствующем архиве.

Система тестирования в текущем состоянии позволяет автоматически решать следующие задачи:

- вести базы для накопления данных о разрабатываемых программах, их версиях, планах тестирования, тестовых и эталонных данных, выполненных корректировках и так далее;
- автоматически выявлять большинство ошибок, обусловленных нарушениями формализованных правил структурного построения модулей и использования данных;
- планировать тестирование и подготавливать рекомендации по систематическому применению различных методов тестирования, чтобы достичь максимальной эффективности и надежности программ в условиях ограниченных ресурсов, выделяемых на тестирование;
- оценивать достигнутую надежность и качество программ по выбранным критериям и определять их конструктивные показатели, такие как логическая и информационная сложность, длительность счета и другие;
- регистрировать изменения в тестируемых программах, вести учет версий программ, в которых проведены те или иные корректировки.

4. Выпуск версии программного комплекса

Использованная в проекте система управления версиями иллюстрируется на Рис.3. Она основана на технологии срезов [4,5], позволяющей создавать независимые ответвления (срезы) исходных текстов проекта, которые, являясь точной копией проекта в момент их образования, далее существуют обособленно – правки в основной части проекта и срезе независимы.

Развитие проекта идет в стволе. Текущие правки вносятся с проверкой на оперативном пакете, для которого по минимуму обеспечивается необходимая надежность. Пользователи же всегда имеют дело с компонентами проекта, собранными из исходных текстов среза, внесению правок в который предшествует весьма обширное тестирование на пользовательских приложениях. При таком подходе срез как раз и содержит тот вариант программного продукта, который называется

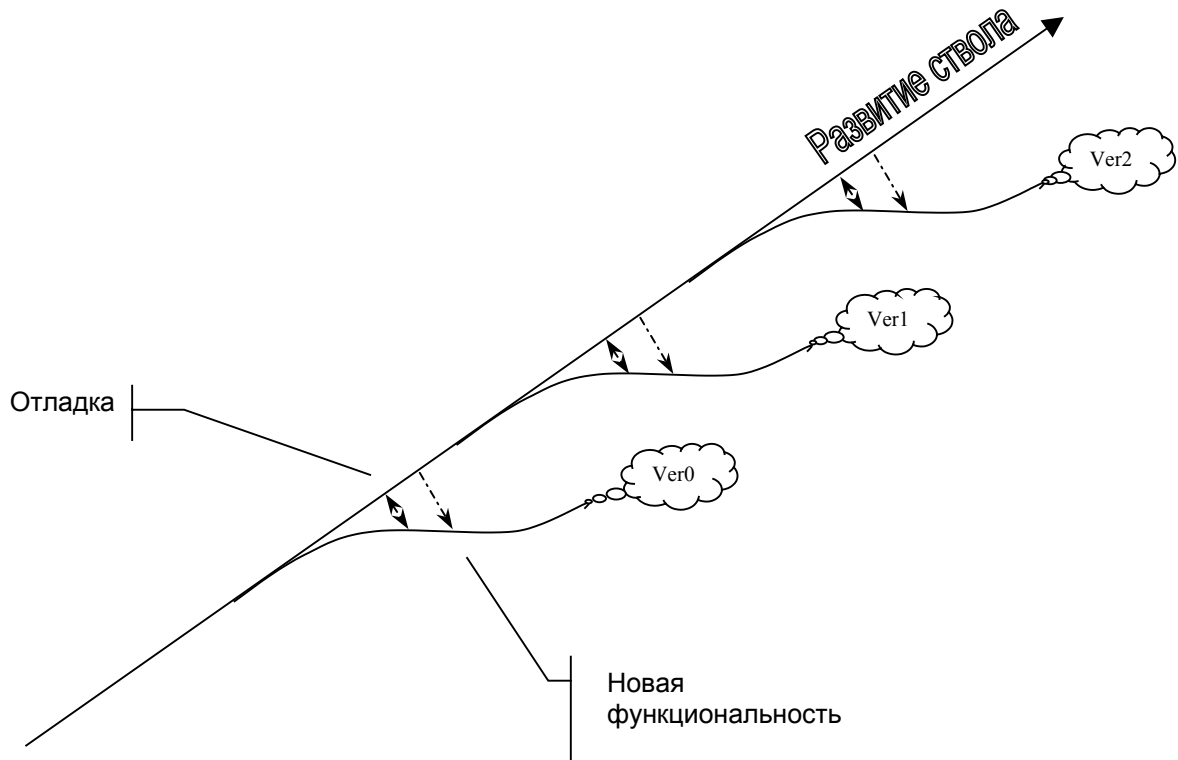


Рис. 3. Система управления версиями с явным разделением текстов

«версией». Соответственно, содержание тестирования, предшествующее внесению в срез, определяется в точности набором требований, предъявляемых к данной версии продукта.

Решение о создании среза принимается, когда в проекте появляется какая-то новая функциональность, критичная для пользователей. К пользователям она попадает с большими временными задержками, обусловленными затратами ресурсов на доводку версии проекта, к ним относятся:

- анализ необходимости создания среза (т.е. выявление новой функциональности, анализ критичности ее отсутствия для пользователей);
- собственно создание среза;
- доведение его уровня надежности от некоторого минимального (стволового) уровня R1 до заданного (определенного для версии) уровня R2, который фиксируется в так называемом протоколе среза (это может быть, например, некоторый набор пользовательских приложений, который должен успешно работать на версии).

Кроме временных проблем, при таком подходе возникают и другие. Во-первых, фактически удваивается объем работы группы контроля качества, поскольку ошибки приходится анализировать как в стволе, так и во вновь созданном срезе. Во-вторых, удваивается и объем работы программистов, так как практически каждую ошибку, исправленную в срезе, приходится править и в стволе. Наконец, со временем срез все больше расходится по исходным текстам со стволом, вследствие чего его сопровождение обходится все дороже, а неудовлетворенность пользователей отсутствием появляющейся в стволе новой функциональности растет.

В силу перечисленных проблем, усугубляющихся при расширении проекта, была предпринята попытка организовать управление надежностью проекта без использования технологий, базирующихся на явном разделении исходных текстов (Рис.4). Заданный уровень надежности было пред-

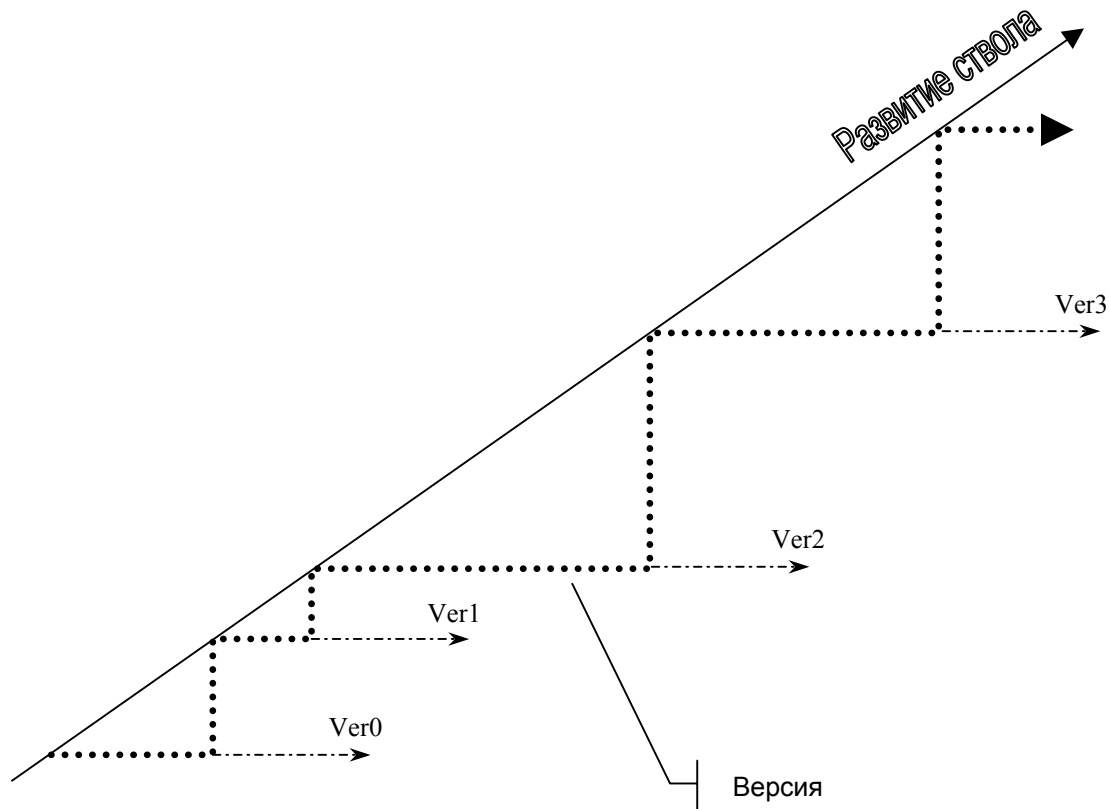


Рис. 4. Система управления версиями без разделения текстов

ложено поддерживать непосредственно в стволе. Любая новая функциональность, которая может снизить надежность, вносится в ствол как опция. Это понятие трактуется в самом широком смысле - опция командной строки запуска компонента проекта, опция параметра сборки и тому подобное.

Вынесение любого развития в опции позволило более аккуратно вводить новую функциональность в основную версию проекта. Можно отлаживать ее как опцию, а включать в основной режим, в котором опции отсутствуют, уже после завершения отладки.

Пусть V – некоторое базовое состояние проекта, рассматриваемое как версия. Оно поддерживается посредством отладки на таком уровне, который обеспечивает выполнение всех требований, предъявляемых к версии. Применительно к версии запрещаются любые правки кроме тех, что устраняют выявленные при отладке ошибки, т.е. для состояния V запрещается какое бы то ни было развитие. Если представить состояние проекта в целом в виде функции $S(o_1, o_2, \dots, o_N)$, где $o_1 \dots o_N$ – это бинарные флаги, активизирующие вновь реализованную функциональность, то базовое состояние проекта V есть $S(0, 0, \dots, 0)$.

Постоянная активность функциональности F_i , задаваемой флагом o_i , достигается при проходе следующих этапов:

- исходное состояние проекта: версия $V = S(0, \dots, o_i = 0, \dots, 0)$;
- рассматривается переход проекта в состояние $V' = S(0, \dots, o_i = 1, \dots, 0)$;
- надежность проекта в состоянии V' доводится до уровня надежности версии R_2 , которому отвечает V ;
- в новом состоянии функциональность F_i активируется по умолчанию, соответствующий ей флаг исключается; состояние V' становится базовым, то есть $V' = S(0, \dots, 0)$.

Как следствие, возрастают требования к оперативному пакету ствола, поскольку возникает естественное желание приблизить уровень надежности R1 оперативного пакета к версионному R2. При этом к нему по-прежнему предъявляются жесткие временные ограничения, но он должен обеспечивать отсутствие деградации работоспособности пользовательских приложений в базовом состоянии проекта В (реально работоспособность которых мы можем проверить лишь в регрессионном тестировании).

5. Формирование оперативного пакета

В качестве основного инструмента для формирования оперативного пакета, отвечающего поставленным требованиям, был выбран анализ покрытия компонентов проекта. Его целью является построение тестового набора, обеспечивающего наиболее полное структурное покрытие кода. В этом случае качество тестов оценивается числом структурных единиц кода, полностью покрываемых данным тестовым набором (метрикой покрытия) [6].

Большинство метрик покрытия строится путем подсчета числа операций, переходов или путей, задействованных при исполнении данного тестового набора. В общем случае полное покрытие обеспечить весьма сложно, а иногда и невозможно, поскольку в программе могут существовать операции или пути как вовсе недостижимые из-за некорректности кода, так и достижимые лишь на специфических негативных тестах. Поэтому метрики являются своего рода мерой качества и тестового набора, и тестируемой программы.

Обычно при анализе покрытия решаются следующие задачи [7]:

- выявление направлений развития тестовых комплектов, обеспечивающих наиболее полное покрытие кода компилятора;

- ликвидация избыточности тестовых комплектов;

- определение метрик покрытия, которые косвенно являются мерами качества тестового пакета.

Сам анализ выполняется путем добавления в анализируемый код специальных счетчиков, например, с помощью утилиты `gsov` [8]. В результате работы такого кода можно получить профиль, содержащий информацию о количестве посещений того или иного структурного элемента, входящего в исходный текст компонента, такого как базовый блок или дуга управления.

Ниже комментируется применение этой технологии в рассматриваемом проекте.

Во-первых, чтобы выявить направление развития оперативного пакета, был реализован ряд утилит, которые позволяют оценить разницу покрытия между регрессионными и оперативными пакетами. Пусть оперативный пакет T1 обеспечивает покрытие $\text{cov}(T1)$, представляющее собой вектор, компоненты которого характеризуют покрытие структурных элементов исходного текста проекта (если пронумеровать все линейные участки, дуги и другие элементы, то i -й компонент вектора определяется как количество посещений соответствующего структурного элемента кода). Предполагается, что регрессионный пакет T2, соответственно, обеспечивает покрытие $\text{cov}(T2)$. Тогда имеется возможность рассмотреть разность покрытий $\text{cov}(T2) \setminus \text{cov}(T1)$, элемент которой с индексом i равен $(\text{cov}(T2)_i - \text{cov}(T1)_i)$, если $(\text{cov}(T2)_i - \text{cov}(T1)_i) > 0$, и равен 0, если это условие не выполняется.

В результате получим вектор $\text{cov} := \text{cov}(T2) \setminus \text{cov}(T1)$, который описывает также некоторое покрытие: в точности то, которое обеспечивается регрессионным, но не обеспечивается оперативным пакетом. Тем самым, выявляются те структурные элементы кода проекта, которые не покрываются оперативным пакетом и покрытие которых должно в первую очередь обеспечиваться при пополнении оперативного пакета.

Во-вторых, данная технология позволяет устранить избыточность оперативного пакета (тем самым уменьшив время его прохождения). Пусть $\text{cov}(T) \oplus \text{cov}(t) := \text{cov}(T \cup \{t\})$ - покрытие, обеспечиваемое тестовым набором T при добавлении к нему теста t. Введем отношение строгого частичного порядка на пространстве векторов покрытий, которое определим следующим образом:

$$a < b \stackrel{\text{def}}{\rightarrow} (\forall i (a_i \neq 0 \rightarrow b_i \neq 0)) \& (\exists i (a_i = 0 \& b_i \neq 0)).$$

Тогда всякий новый тест t включается в оперативный пакет $T1$ только в том случае, если выполнено условие избыточности:

$$\text{cov}(T) < \text{cov}(T) \oplus \text{cov}(t),$$

то есть, если тест t обеспечивает покрытие каких-либо структурных элементов кода, которые не покрывались тестовым набором T . Необходимо отметить, что здесь не ставится задача сокращения пакета (хотя она также представляет определенный интерес) - лишь избыточный пакет.

6. Результаты применения подхода

Рассмотрим некоторые результаты. Для их иллюстрации воспользуемся аппаратом метрик, который является одним из наиболее широко применяемых средств анализа в области современных технологий контроля качества.

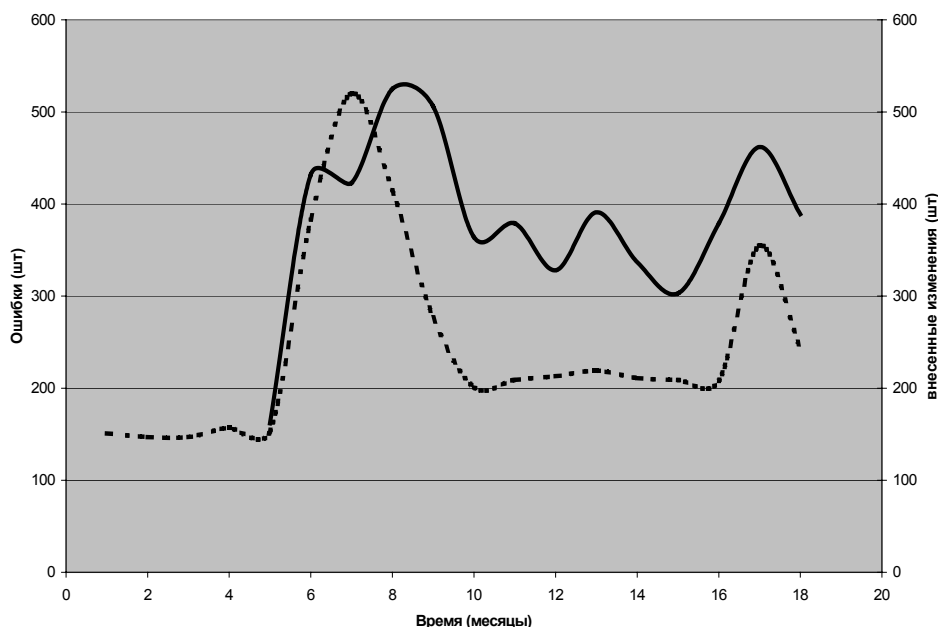


Рис. 5. Динамика обнаружения ошибок и внесение изменений

В случае программной системы одной из таких метрик является количество ошибок, выявляемых в единицу времени. Процесс исправления ошибок и развития проекта характеризуется частотой внесения изменений (количеством модификаций проекта в единицу времени). Первый график (Рис. 5) характеризует совместную динамику этих двух метрик. По горизонтальной оси задано время t в месяцах, отражена история проекта за период около двух лет. Как видно, они неплохо коррелируют. Пик в районе $t=7$ соответствует работе над срезом, пик в районе $t=17$ — работе над выпуском надежной версии без разделения текстов. И в том и в другом случаях ужесточение требований к продукту приводит к росту как числа ошибок, так и числа правок.

На следующем графике (Рис. 6) наряду с количеством ошибок, выявленных в единицу времени, задана также средняя продолжительность жизненного цикла ошибки - в некотором смысле, эта метрика характеризует трудоемкость отладки. Как видно, при использовании среза рост количества ошибок хорошо коррелирует с трудоемкостью отладки, при отказе от разделения текстов ($t=17$) трудоемкость, напротив, резко падает.

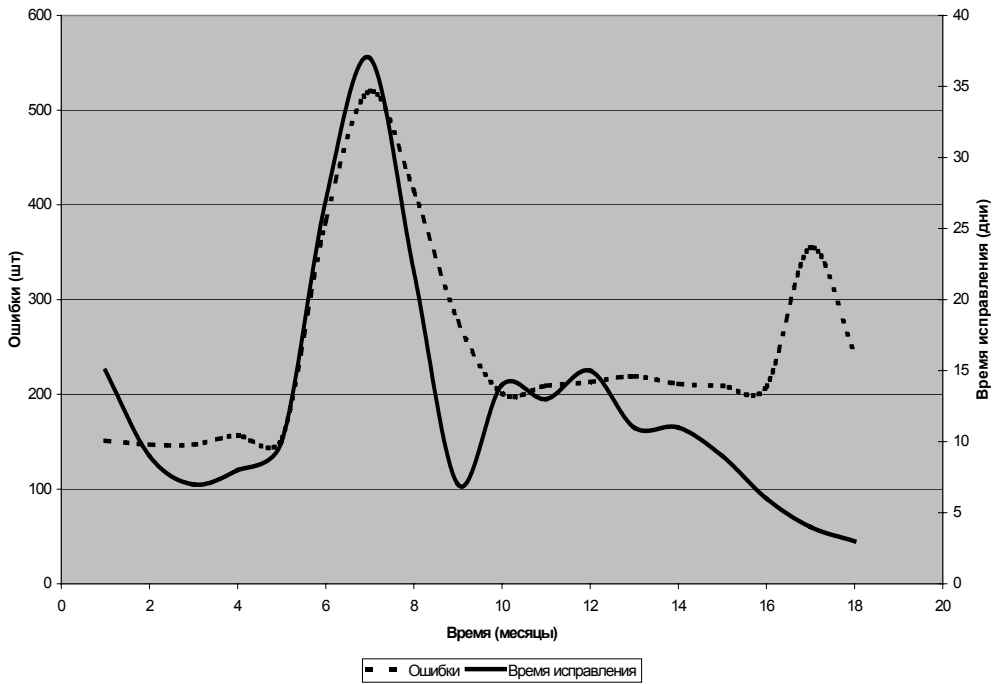


Рис. 6. Динамика обнаружения и исправления ошибок

Наконец, последний график (Рис. 7) демонстрирует рост качества продукта, что свидетельствует об успешности примененного подхода. На этом графике данные приведены за период в течение последнего месяца работы над проектом (время задано в днях; в терминах предыдущих двух графиков это период $t=17-18$).

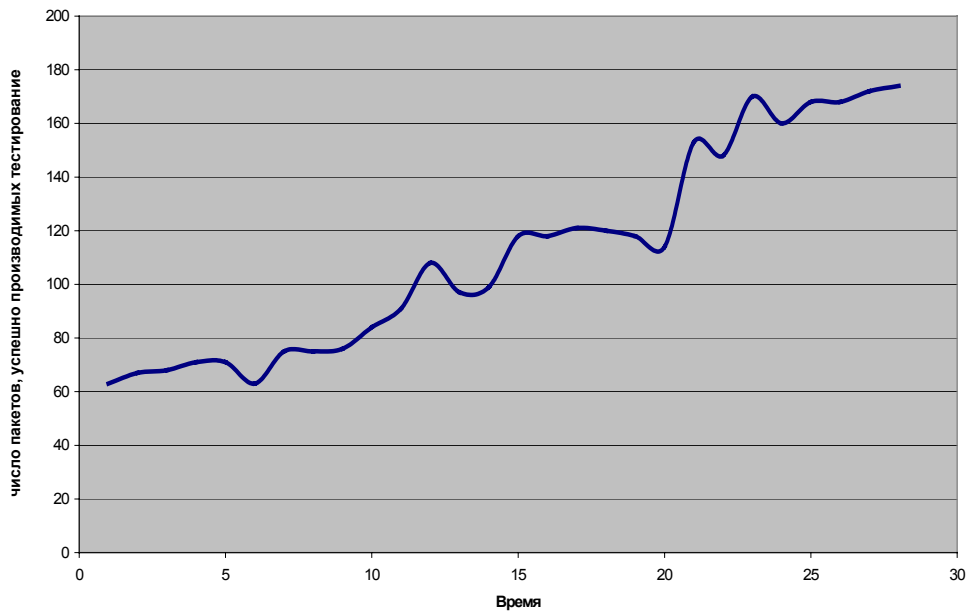


Рис. 7. Динамика изменения качества продукции

Необходимо заметить, что проанализированный временной отрезок разработки проекта пришелся на период особенно бурного его развития и активного наполнения новой функциональностью.

Приведенные результаты свидетельствуют о перспективности выбранного подхода и оправданности дальнейших исследований в этом направлении.

Заключение

В статье рассмотрены некоторые аспекты организации процесса тестирования компиляторного проекта. Затронуты вопросы построения комплекса программных средств тестирования, а также подходы к организации самого процесса разработки и тестирования проекта и различные критерии оценки этого процесса.

Основным результатом, полученным авторами при работе по данной тематике, можно считать реализацию ряда мер, направленных на оптимизацию системы обеспечения качества компиляторных проектов

С точки зрения дальнейшей работы представляет интерес большая интеграция с прочими подсистемами, обеспечивающими разработку программного комплекса, например, с БД регистрации ошибок, выявляемых в программных продуктах в процессе отладки. Перспективной темой является также встраивание различного рода автоматических анализаторов, решающих рутинные проблемы отладки.

Еще одним направлением дальнейших исследований представляется более широкое использование возможностей структурного анализа исходных текстов, частным случаем которого можно считать описанную в статье технику покрытия. В частности, интересно рассмотреть применение такого рода анализа для выявления зависимости тех или иных компонентов проекта, что в определенных случаях позволяет вводить эвристики, серьезно сокращающие объем тестирования.

Литература

1. Ф. Брукс, Мифический человеко-месяц или как создаются программные системы. СПб.: Символ-Плюс, 2001.
2. Роберт Калбертсон, Крис Браун, Гэри Кобб. Быстрое тестирование, М., Издательский дом Вильямс, 2002.
3. R.Savoie. The DejaGnu Testing Framework for dejagnu Version 1.3, 1996 (<http://www.gnu.org/software/dejagnu/>).
4. Ю.В. Баскаков, А.А. Лаврешников, Р.Ю. Рогов, Л.Г. Тарасенко, Е.Ю. Чернова. Вопросы организации систем обеспечения качества оптимизирующих компиляторов. В сб. трудов ИМВС РАН. М.: ИМВС РАН, 2004.
5. Введение в CVS. <http://www.citforum.ru/programming/digest/cvsintrotorus.shtml>
6. <http://www.ldra.co.uk/pages/metrics/ter1.htm> — каталог метрик покрытия.
7. C-Cover — Software Test Coverage Analysis. Технический обзор. Codework Ltd., Великобритания, 2001 (<http://www.codework.com/c-cover/detailed.htm>).
8. M. Harder et al., Improving Test Suites via Operational Abstraction. Proceedings of the 25th international conference on Software engineering, IEEE Computer Society, Washington, DC, USA, 2003, pp. 60–71.

Лаврешников Андрей Александрович. Родился в 1977 году. В 2000 году окончил МАТИ им. К.Э. Циолковского. Автор 11 научных трудов. Область научных интересов: верификация и тестирование компиляторов. Старший научный сотрудник ЗАО МЦСТ.

Рогов Роман Юрьевич. Родился в 1977 году. В 2000 году окончил МГУ им. М.В. Ломоносова. Автор 8 научных трудов. Область научных интересов: открытые системы, верификация и тестирование компиляторов. Ведущий научный сотрудник ЗАО МЦСТ.