ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 32 за 2018 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Бобков В.Г.

Разработка и автоматическое регрессионное тестирование программного комплекса NOISEtte

Рекомендуемая форма библиографической ссылки: Бобков В.Г. Разработка и автоматическое регрессионное тестирование программного комплекса NOISEtte // Препринты ИПМ им. М.В.Келдыша. 2018. № 32. 20 с. doi:10.20948/prepr-2018-32
URL: http://library.keldysh.ru/preprint.asp?id=2018-32

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.КЕЛДЫША
Р о с с и й с к о й а к а д е м и и н а у к

В. Г. Бобков

Разработка и автоматическое регрессионное тестирование программного комплекса NOISEtte

Москва — 2018

Бобков В. Г.

Разработка и автоматическое регрессионное тестирование программного комплекса NOISEtte

Рассмотрены алгоритмы и подходы, использованные при разработке системы автоматического регрессионного тестирования программного комплекса NOISEtte, разрабатываемого в Институте прикладной математики им. М.В. Келдыша РАН. Изложены проблемы и особенности совместной разработки сложного программного кода на основе системы контроля версий, а также использования ее в процессе тестирования программного пакета.

Ключевые слова: разработка кода, регрессионное тестирование, совместная разработка

Vladimir Georgievich Bobkov

NOISEtte CFD package development and regression testing system

The CFD package NOISEtte developed within CAA laboratory of KIAM RAS and automated regression testing system design and features are considered. Features of using revision control system in the CFD software development process and in testing system are outlined.

Key words: software development, regression testing, version control system

Оглавление

Введение

При реализации численных алгоритмов в составе сложного универсального, расчитанного на решение широкого круга задач программного комплекса (ПК), в разработке которого задействована команда из нескольких специалистов, рано или поздно приходится начинать использование каких-либо средств совместной разработки. Наиболее популярными решениями среди программистов на протяжении последних 10-20 лет являются системы совместной разработки на основе систем контроля версий (SVS – Software Versioning System) такие как `CVS`[1] `(Concurent Versions System)` [1], `SVN(Subversion)` [2], `Mecurial` [3], `Git` [4].

Использование таких систем существенно упрощает совместную разработку программного продукта, обеспечивая сохранность целостности кода и централизованное (или децентрализованное) хранение полной истории изменений, но ответственность за сохранение корректной функциональности кода все равно лежит всецело на плечах разработчиков.

При разработке код непрерывно модифицируется — в него вносятся изменения, реализующие новую функциональность и отражающие модернизацию алгоритмов и исправления ошибок. Этот процесс может приводить к непредусмотренному изменению поведения программы в различных рабочих сценариях. Чтобы этого избежать, применяется **регрессионное тестирование** — набор тестов и методика тестирования, направленные на проверку работоспособности работающей ранее функциональности программы.

Целью применения систем автоматического регрессионного тестирования является частичный перенос ответственности за поддержку кода в рабочем состоянии с разработчиков на сторону автоматизированных систем. Применение автоматического регрессионного тестирования наиболее актуально при поддержке и интенсивной работе над кодом ПК большого объема с большим числом разработчиков. В такой ситуации один разработчик, как правило, обладает знанием определенной части кода, но не обладает достаточным знанием об остальной части кода и о связях между модулями внутри всего кода в целом и, как следствие, не имеет возможности спрогнозировать, как изменения кода одного модуля повлияют на работоспособность всего программного комплекса. К тому же в силу того, что различных сценариев работы программного продукта может быть огромное количество, простое тестирование работоспособности программы на нескольких типичных сценариях после внесения изменений в код не является гарантией корректной работы при использовании других сценариев.

Далее будет изложены особенности совместной разработки и использо-

[1]Разработка CVS была завершена в 2008 году и постепенно вытесняется более современными системами

вания системы автоматического регрессионного тестирования программного комплекса NOISEtte, разрабатываемого в секторе вычислительной аэроакустики Института прикладной математики им. М.В. Келдыша РАН.

Заметим, что в данной статье освещается не функциональность ПК NOISEtte, о которой можно получить информацию в работе [5], но средства разработки и тестирования, используемые при работе над ним. Также далее под терминами «тест» и «тестирование» понимаются именно регрессионный тест и регрессионное тестирование.

Совместная разработка кода

Как уже было упомянуто выше, в современном программировании невозможно представить развитие сложного объемного проекта без использования системы совместной разработки. Здесь и далее под такой системой будет подразумеваться `SVN(Subversion)`, хотя все нижесказанное применимо и для других систем совместной разработки.

Использование системы совместной разработки на основе контроля версий существенно упрощает работу над программным кодом командой программистов во многих аспектах, обеспечивая:

- возможность бесконфликтной одновременной удаленной работы над единым кодом множества разработчиков;
- мгновенную доступность изменений кода всему коллективу разработчиков;
- возможность оперативной отмены ошибочных изменений и возврат к работоспособной версии кода;
- возможность ведения одновременной параллельной разработки альтернативных версий («ветвей») кода для разработки новой функциональности программного продукта без ущерба для основной ветки разработки;
- возможность слияния веток и ревизий;
- централизованное хранение кода, существенно упрощающее резервное копирование кода проекта.

Это лишь малая часть возможностей, доступных при использовании системы контроля версий.

Общепринятой практикой при разработке крупных программных продуктов несколькими специалистами является использование синтеза технологий систем контроля версий с системой управления проектами и (или) с системами отслеживания ошибок (bugtracking) с интегрированными средствами сопровождения кода, такими как `Redmine` [6], `Trac` [7], `JIRA` [8], `Basecamp` [9] и др. Интеграция такого инструмента с системой контроля версий позволяет связать планирование развития кода непосредственно с изменениями в коде. Далее, не ограничивая общности, будем рассматривать схему, используемую при разра-

ботке ПК NOISEtte — систему контроля версий Subversion (далее – SVN) и систему планирования и отслеживания ошибок Redmine.

Redmine — достаточно гибкая система, допускающая детальную настройку схемы возможных состояний (так называемых «статусов») задачи и допустимых переходов между ними. При этом вся работа по задаче ведется с использованием веб-интерфейса, доступного разработчикам как по локальной сети, так и по сети интернет при должной настройке и наличии доступа извне к серверу, на котором установлена система.

Типичный цикл разработки и изменение статуса задачи при использовании такой схемы в рамках системы разработки ПК NOISEtte выглядит следующим образом (см. рис. 1). В системе планирования разработчик создаёт задачу о необходимости реализации новой функциональности ПК, и ей присваивается идентификатор, статус задачи при этом автоматически принимает значение «Новая». Далее назначается ответственный за разработку этого задания, по возможности устанавливаются сроки ее выполнения и далее, при изменениях кода, все изменения, помеченные этим идентификатором, автоматически помещаются в историю соответствующей задачи. При этом статус задачи выставляется в «Процесс идет» или, в случае если по той или иной причине решение задачи признано нецелесообразным или невозможным, статус задачи меняется на «Отказ» и задача закрывается. Далее, при необходимости, создаются и решаются задачи, без которых невозможно завершение текущей, — так называемые «блокирующие» задачи. Статус текущей задачи во время решения блокирующих задач выставляется в «Заблокирована». По завершении блокирующих за-

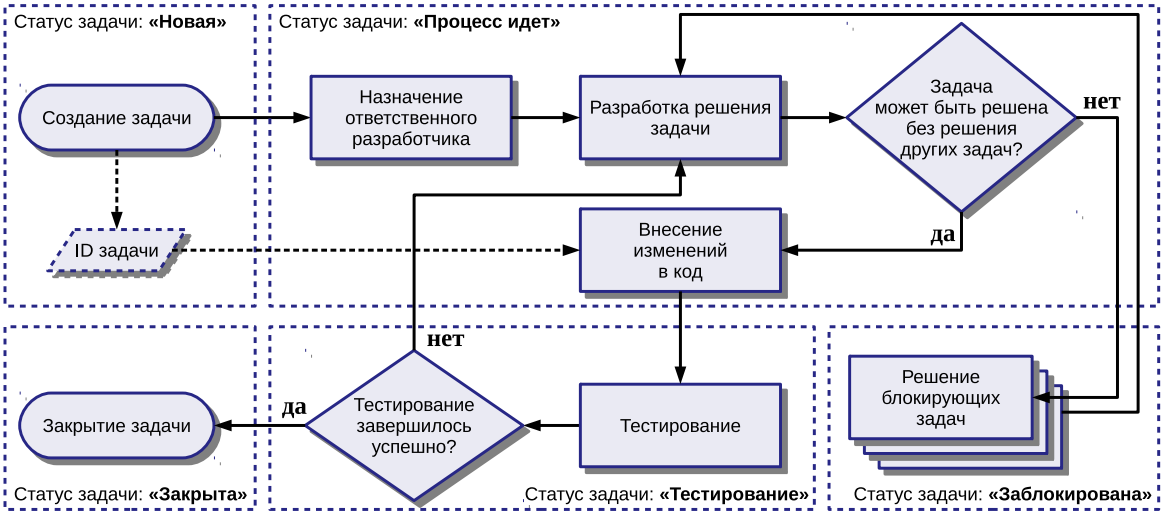

Рис. 1. Схема жизненного цикла задачи в системе Redmine

дач и внесении в код изменений, реализующих решение исходной задачи, ей присваивается статус «Завершена», или «Тестирование» – в случае если реализованную функциональность необходимо тестировать. После тестирования, в

зависимости от его результата, задача закрывается (статус выставляется в «Закрыта») или возвращается разработчику на доработку (статус задачи при этом снова принимает значение «Процесс идет», и при необходимости корректируются сроки выполнения задачи).

Процесс исправления ошибок в случае обнаружения ошибочной работы программы выглядит аналогично с тем отличием, что обнаруживший ее пользователь или разработчик при создании задачи подробно описывает проблему, ревизию кода, в которой обнаружена ошибка, и описывает последовательность действий или прилагает набор данных, необходимых, чтобы воспроизвести ошибку. Далее определяется, в чьей компетенции находится ошибочный код, задача переводится ответственному разработчику, и после этого процесс идет по описанной выше схеме. По завершении решения проблемы задача возвращается открывшему ее лицу для тестирования и подтверждения или опровержения исправления ошибки и закрытия задачи или возврата ее на доработку.

Как правило, данные и задача, на которых была воспроизведена ошибка, в последующем используются для создания тестов, если это возможно.

Поддержание научно-исследовательского кода в рабочем состоянии и его тестирование

Главным отличием научно-исследовательского кода от промышленного и коммерческого кода заключается в том, что в нем непрерывно реализуются новые методы и алгоритмы, свойства которых необходимо исследовать. В отличие от промышленных программных комплексов, набор базовых алгоритмов («ядро») которых меняется крайне редко, и, как следствие, для проверки корректности работы ядра достаточно иметь однажды сформированный фиксированный набор тестов, состав методов, реализованных в научно-исследовательском коде, постоянно эволюционирует, обновляется и пополняется новыми реализациями. Соответственно, и база тестов должна постоянно обновляться и пополняться тестами для новых методик.

При реализации новых алгоритмов или новых возможностей в ПК хорошим тоном, помимо документирования, считается создание набора тестов, позволяющих проверить корректность работы новой функциональности. Такой подход обеспечивает широкое покрытие всех доступных сценариев использования ПК.

Рассмотренный в данной статье программный комплекс NOISEtte является научно-исследовательским развивающимся кодом с постоянно обновляющимся набором тестов. Под «тестом» здесь и далее понимается сценарий запуска ПК и утилит, входящих в его состав, и получение некоторого анализа результата после отработки этого сценария, позволяющего однозначно судить о том, успешно прошел тест или нет.

Как уже было упомянуто выше, при разработке научно-исследовательского

кода неизбежно наступает момент, когда ранее корректно работавший набор тестов становится некорректным из-за ошибок в реализации новой функциональности, из-за непредвиденных эффектов взаимодействия модифицированного кода с остальной инфраструктурой ПК или из-за ошибок во вновь созданных тестах.

Тесты ПК NOISEtte

Используемое здесь понятие «тест» как сценарий запуска ПК и составляющих его утилит с последующим анализом результата предоставляет полную свободу в конструировании тестов. На данный момент среди тестов ПК NOISEtte можно выделить два основных типа.

Первый — типичный тест для программных комплексов, предназначенных для решения задач численного моделирования, представляет собой постановку небольшой задачи с численным решением, полученным с помощью рабочей версии кода ПК. Далее это численное решение считается эталонным. Сценарий теста включает в себя этап решения задачи средствами ПК с последующей проверкой соответствия нового численного решения эталонному. Анализ результата такого теста заключается как в проверке самого факта успешного безавостного решения задачи, так и в сравнении разницы между численным и эталонным решениями в какой-либо норме с наперед заданным достаточно малым значением. Примером такого теста может служить, например, решение задачи о распространении возмущения в ограниченной области. Этот тест может быть поставлен во множестве вариаций — на разных расчетных сетках, с разными способами заданными возмущениями, с разными базовыми моделями и так далее, то есть он может охватить достаточно широкий спектр функциональности ПК. При этом нет необходимости считать полностью задачу для всего временного интервала задачи, чтобы судить об изменении характера решения. Достаточно начать расчет с заранее сохраненной точки восстановления и произвести расчет на небольшом интервале времени. Дальнейшее сравнение полученного решения с эталонным, взятым на этот же момент времени, позволит судить о том, произошла ли деградация кода.

Тесты второго типа предназначены для проверки корректной функциональности последовательности вызовов, имитирующих некоторые типовые сценарии работы ПК и входящих в его состав утилит. Тесты такого типа позволяют в одном сценарии проверить работу нескольких утилит, входящих в состав ПК. Условием успешного выпонения теста, помимо получения ожидаемого результата на выходе, является корректная работа всех утилит, использованных в сценарии. Примером может служить, например, тест, проверяющий функциональность подготовки сетки к расчету. В этом тесте последовательно вызываются утилита преобразования расчетной сетки из стороннего формата в формат

NOISEtte, утилита преобразования сетки в бинарный формат, утилита достраивания периодических граничных условий и утилита проверки корректности сетки.

Система автоматического тестирования ПК NOISEtte: требования

Пока число тестов ПК NOISEtte было невелико и покрытие ими кода было весьма поверхностным, они запускались в ручном режиме с нерегламентированной периодичностью. Однако, когда покрытие стало улучшаться и число тестов превысило 200, стало очевидным, что разработку, настройку и запуск тестов необходимо систематизировать и по возможности автоматизировать.

Как было указано выше, на момент разработки системы имелось более 200 тестов первого типа, оформленных в виде отдельных задач, в которых использовался специально разработанный в пакете “режим тестирования”, в котором по завершении решения задачи проводилось сравнение полученного численного решения с рассчитанным ранее эталонным численным решением. Каждый тест располагался на диске в отдельном каталоге, а исполнение тестов выглядело как запуск программы на скриптовом языке, последовательно выполняющей запуск ПК NOISEtte в очередном каталоге (на очередной тестовой задаче). Были написаны отдельные программы запуска и генерации отчета для Windows (batch-файл) и для Linux (shell-script).

Когда наступала необходимость провести тестирование, разработчик компилировал текущую версию ПК, брал текущий набор тестов и запускал в корневом каталоге соответствующую целевой платформе программу-скрипт. По завершении выполнения всех тестов на выходе разработчик получал текстовый файл-отчет о результатах прохождения каждого теста в виде «прошел/не прошел».

Такая методика тестирования была признана неудобной, и было решено заменить ее более изящной системой. Новая система автоматического тестирования была построена на основе следующих требований.

1. Структура хранения тестов должна быть сохранена — каждый тест в отдельном каталоге.
2. Набор тестов должен быть связан с версией кода, на которой этот набор отрабатывает корректно.
3. Разработчику должна быть предоставлена возможность в любой момент времени узнать подробную информацию о результате тестирования текущей версии кода.
4. Разработчику должна быть предоставлена возможность самому выполнять отдельные тесты с использованием своей версии исполняемых фалов ПК и анализировать результаты их исполнения на платформах Windows

и Linux.

5. Если изменения разработчика привели к деградации кода (то есть не все тесты проходят корректно), то он должен быть извещен об этом автоматически.

На основе этих требований была разработана система автоматического тестирования ПК NOISEtte.

Система автоматического тестирования ПК NOISEtte: внутренняя структура

Требования о сохранении структуры хранения тестов и о связывании кода и тестов были удовлетворены путем переноса существующей структуры хранения тестов в SVN-хранилище, содержащее исходный код программного комплекса. Результатом такого перемещения было присвоение единой ревизии согласованной паре «код–тесты». Под «согласованной» парой здесь понимается исходный код ПК и такой набор тестов с параметрами, что с использованием исполняемых файлов пакета программ и утилит, скомпилированных из этой ревизии кода, весь набор тестов успешно выполняется. Иными словами, согласованность пары «код–тесты» является подтверждением корректной работоспособности ПК и отсутствия деградации кода. Очевидно, что из-за постоянно изменяющегося кода программ не каждая последующая ревизия и соответствующая ей пара «код–тесты» будет согласованной. Собственно, в этом и заключается основная цель разработки системы тестирования — определить и сократить период существования несогласованной пары или, другими словами, период существования некорректно работающего кода.

Для удовлетворения требования о возможности самостоятельного запуска разработчиком отдельных тестов под разными целевыми платформами решено было основную единую управляющую программу с функциональностью запуска отдельных тестов разработать на интерпретируемом языке Python [10], который сочетает простоту, кроссплатформенность и независимость от наличия каких-либо компиляторов. Помимо этого, существует множество дополнительных библиотек-модулей, позволяющих существенно расширить возможности языка.

В основном автоматическом режиме программа извлекает из SVN текущую версию кода и компилирует все необходимые бинарные файлы, входящие в состав ПК, извлекает из хранилища текущий набор тестов, с использованием этих бинарных файлов выполняет все тесты, обрабатывает результаты и генерирует полный отчет о результатах тестирования и оповещает разработчиков в случае деградации кода. Второй — «ручной» режим – аналогичен первому, но в нем предполагается, что исполняемые файлы и состав тестового набора предоставляются программе пользователем, а оповещения не рассылаются. Алгоритми-

ческая схема управляющей программы представлена на рис. 2.

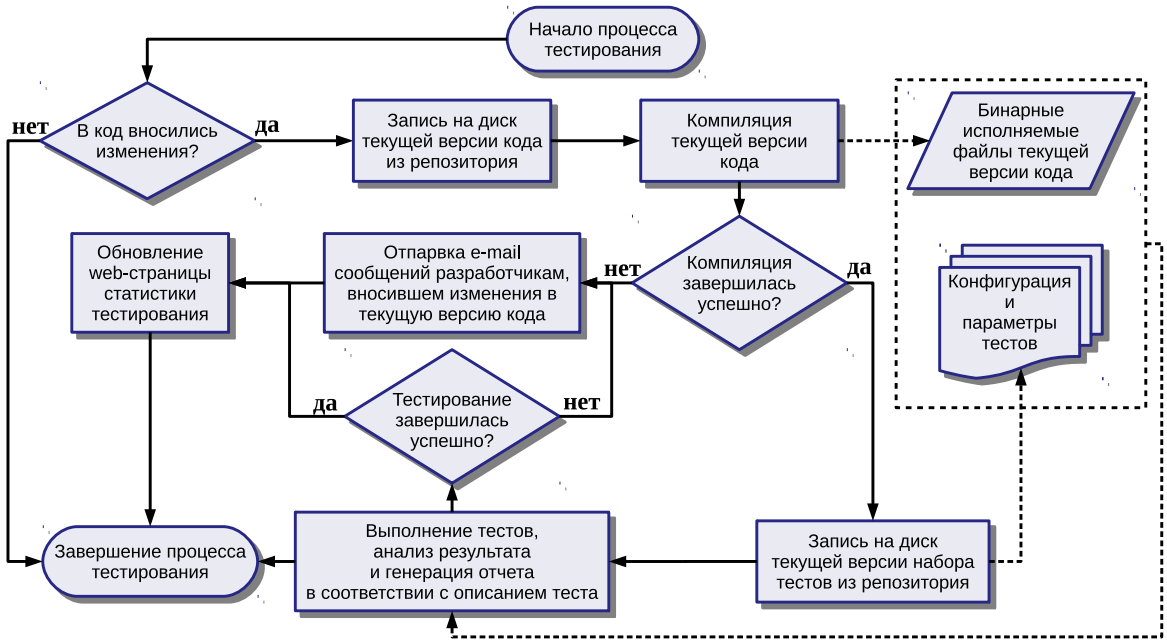

Рис. 2. Схема управляющей программы системы автоматического тестирования

Схема предусматривает следующую последовательность действий. Вначале программа проверяет, были ли в код внесены изменения со времени последнего запуска тестирования, и процесс тестирования продолжается, только если код изменялся. Из SVN извлекается текущая версия кода ПК и производится попытка его скомпилировать. Если компиляция завершилась неуспешно, процесс тестирования прерывается и разработчикам, изменявшим код со времени последнего успешного тестирования, отсылаются оповещения по электронной почте. В случае успешной компиляции программа извлекает из SVN текущий набор тестов и производит последовательное выполнение каждого теста, заполняя при этом отчет о ходе тестирования. По окончании тестирования программа в случае наличия ошибок в ходе выполнения тестов отправляет оповещения по электронной почте о деградации кода разработчикам, изменявшим код со времени последнего успешного тестирования.

Реализация функциональности запуска теста и анализа результатов запуска была наиболее сложным этапом разработки управляющей программы системы тестирования.

Система автоматического тестирования ПК NOISEtte: сценарий теста

Для каждого теста было решено объединить его описание, сценарий его выполнения и сформулировать набор условий успешного или неуспешного за-

вершения в один конфигурационный файл, написанный на расширяемом языке разметки XML [11]. Этот формат был выбран за простоту его формального синтаксиса, гибкие возможности формирования документов и наличие в языке Python библиотек для его разбора.

Разберем подробно содержание конфигурационного файла теста на конкретном примере конфигурационного файла реального теста из существующего набора ПК NOISEtte – см. Листинг 1.

В конфигурационном файле присутствует один корневой элемент и атрибуты, исчерпывающе описывающие тест, сценарии его запуска и условия проверки корректности его выполнения.

Файл в соответствии со стандартом языка начинается с пролога, в котором указаны версия языка (1.0) и следующие элементы и атрибуты.

testcase — единственный корневой элемент с атрибутом “name”, содержащим уникальное имя теста. Включает в себя дерево элементов, собственно определяющих сценарий теста.

description — элемент с единственным атрибутом “text”, содержащим текстовое описание теста. Входит в состав элемента testcase.

run — элемент, описывающий один сценарий запуска теста. Содержит атрибуты “name” – название сценария, “stdout”, “stderr” – имена файлов, в которые перенаправляется вывод всех команд, вызываемых в сценарии. Набор таких элементов входит в состав элемента testcase и определяет набор сценариев теста.

call — элемент, описывающий вызов одной команды или исполняемого файла в сценарии. Содержит атрибуты “cmd” – имя исполняемого файла, “path” – опциональный параметр пути к исполняемому файлу (если не указан, то при вызове используются системные пути, специфичные для текущей платформы), “description” – описание команды, “stdout”, “stderr” – опциональные имена файлов, в которые перенаправляется вывод команд (если не указаны, то вывод перенаправляется в соответствующие файлы сценария). Также опционально элемент call может содержать элемент prefix с аргументами “cmd” и “arguments”, определяющими, какая команда и с какими параметрами должна быть добавлена перед вызовом текущей команды. Функциональность элемента prefix задействуется, когда необходимо вызывать бинарные выполняемые файлы не напрямую, а через какие-либо прослойки – например, через mpirun при необходимости использования MPI. Набор элементов call входит в состав элемента run и определяет последовательность команд и вызовов, происходящих в процессе выполнения сценария.

argument — элемент, определяющий аргумент вызова команды или исполняемого файла в сценарии. Содержит атрибуты “name” – имя аргумента, и

```
<?xml version="1.0"?>
<testcase name="QA_UTILS approx">
  <info text="approx flow testcase ..."/>
  <run name="SPMPI" stdout="testcase.stdout" stderr="testcase.
     stdout">
    <call cmd="specutil.x" path="$NOISETTE_SANDBOX/bin" description
       ="mesh generation" stdout="genmesh.std" stderr="genmesh.std"
       >
      <argument name="util" value="genmesh"/>
      <argument name="params" value="P MESH/IN 6x6x6"/>
      <prefix cmd="mpirun" arguments="-np 4"/>
    </call>
    <call cmd="meshperiodic.x" path="$NOISETTE_SANDBOX/bin"
       description="making periodics" stdout="meshperiodic.std"
       stderr="meshperiodic.std">
      <argument name="whole" value="2 4 MESH/IN MESH/PER -X=0 -Y=0
         -Z=0 -L=B -M=40 -N=100000 -CC"/>
    </call>
  </run>
  <passconditions>
    <condition type="contains">
      <file path="specutil_genmesh.std"/>
      <word text="MeshFormatConverter: Finished!"/>
    </condition>
    <condition type="notcontains">
      <file path="specutil_genmesh.std"/>
      <word text="SIGSEGV"/>
      <word text="Epic fail"/>
    </condition>
    <condition type="exists">
      <file path="./MESH/IN/mesh.txt" noresult="1"/>
      <file path="./derivs.dat" noresult="1"/>
    </condition>
    <condition type="check_tecplot_ascii">
      <file path="./derivs.dat" noresult="1"/>
      <field var="A000" what="maxabs" cond="less" val="5e-14"/>
      <field var="A100" what="maxabs" cond="less" val="5e-14"/>
    </condition>
  </passconditions>
  <results>
    <file path="./pex_eaxct_vs_frun.png"/>
  </results>
</testcase>
```

Листинг 1. Пример содержания XML-файла конфигурации теста

“value” – собственно аргумент. Набор таких элементов входит в состав элемента `call` и определяет набор аргументов, с которыми будет произведен вызов родительского элемента `call`.

`passconditions` — элемент, описывающий набор условий, проверка которых после выполнения всех сценариев позволяет судить об успешном или неуспешном завершении теста. Атрибутов не имеет.

`condition` — элемент, описывающий условие, которое необходимо проверить. Имеет единственный аргумент “type” – тип проверяемого условия. Поддерживаются следующие типы условий: “contains”, “notcontains”, “vartol”, “exists”, “check_tecplot_ascii”, “matched”, подробное описание которых приводится ниже.

`file` — элемент, описывающий файл, к которому будет применяться проверка условия. Опциональный аргумент элемента – “noresult”, наличие которого указывает на то, что проверяемый файл не должен включаться в результаты теста. Набор элементов `file` включается в каждый элемент `condition` или отсутствует (тогда условие применяется к файлу, в который сохранялся вывод сценария).

`word` — элемент, определяющий строку, наличие или отсутствие которой в файлах необходимо проверить. Имеет единственный атрибут “text”, собственно, определяющий содержание строки. Наборы таких элементов присутствуют в условиях типа “contains” и “notcontains”.

`variable` — элемент, определяющий переменную, значение которой надо сравнить с указанным значением. Аргументами являются “name” – имя переменной и “tolerance” – значение, определяющее максимально допустимое значение переменной. Наборы таких элементов присутствуют в условии типа “vartol”. При этом предполагается, что значение переменной задано в тексте проверяемого файла в виде пары “имя” и “значение”, разделенных знаком равенства или любым числом пробельных символов.

`field` — элемент, определяющий проверку поля переменной, которое сохранено в текстовый файл данных в формате `Tecplot`. Аргументами элемента являются “var” – имя переменной в файле данных, “what” – операция, производимая над полем переменной, “cond” – условие, которое необходимо проверить, и “val” – значение, используемое при проверки условия. Аргумент операции может принимать значения “min”, “max” и “maxabs”, что соответствует операциям вычисления максимального значения, минимального значения и максимального значения модуля. Аргумент “cond” может принимать значения “less” и “greater”, что означает проверку условия “результат операции над полем” меньше или больше значения заданного аргументом “val”. Наборы элементов `field` присутствуют исключительно в условиях типа “check_tecplot_ascii”.

Помимо описанных типов условий, также реализованы условия типа

“exists” и “matched”, в первом из которых проверяется существование файлов, определенных набором элементов `file`, а во втором проверяется совпадение содержания всех файлов, определенных набором элементов `file`.

`result` — элемент, определяющий, какие файлы должны быть сохранены и включены в результаты тесты для последующего анализа. Включает в себя набор элементов `file`, определенных выше. Заметим, что все файлы вывода, определенные аргументами “stdout” и “stderr” в элементах `file` и `call`, автоматически включаются в результаты теста. Также в результаты теста включаются файлы, используемые при проверке условий, если для них не включен флаг “noresult”. При этом для всех файлов, интерпретируемых как результат теста, перед добавлением их в качестве результата производится проверка, существуют ли они и не пусты ли они — в противном случае они игнорируются. Также при выполнении теста всегда записывается в файл “conditions.txt” подробный отчет о результатах проверки условий. Таким образом, конфигурационный файл, приведенный выше, трактуется программой автоматического тестирования следующим образом.

Выполнить тест “QA_UTILS approx” с одним сценарием “SPMPI”и файлом вывода testcase.stdout, состоящим из двух вызовов:

mpirun -np 4 $NOISETTE_SANDBOX/bin/specutil.x genmesh P MESH/IN 6x6x6 (вывод перенаправить в файл genmesh.std)

$NOISETTE_SANDBOX/bin/meshperiodic.x 2 4 MESH/IN MESH/PER -X=0 -Y=0 -Z=0 -CC (вывод перенаправить в файл meshperiodic.std)

Тест считать завершившимся успешно, если

– файл genmesh.std содержит строку “MeshFormatConverter: Finished!”;

– файл genmesh.std не содержит строку “SIGSEGV” и не содержит строку “Epic fail”;

– файлы ./MESH/IN/mesh.txt и ./derivs.dat существуют (файлы в результаты теста не включать);

– в файле данных ./derivs.dat в формате Tecplot максимум модуля поля переменной A000 меньше значения 5e-14 и максимум модуля поля переменной A100 меньше значения 5e-14 (файл в результаты теста не включать).

В результаты теста включить файлы (если они существуют и не пусты): testcase.stdout, genmesh.std, meshperiodic.std, pex_eaxct_vs_frun.png, conditions.txt.

Система автоматического тестирования ПК NOISEtte: реализация

В программе автоматического тестирования был реализован разбор описанных выше конфигурационных файлов, запуск сценариев в виде последовательности вызовов указанных исполняемых файлов с указанными параметра-

ми, проверка описанных выше условий с последующей генерацией отчетов о выполнении сценариев теста и сохранением результатов.

Для разбора конфигурационного файла использовались утилиты стандартного Python модуля xml.etree.ElementTree, позволяющего последовательно обходить конструкции древовидной структуры XML-файла и извлекать описания элементов и аргументов.

Для запуска исполняемых файлов в соответствии со сценарием использовался функционал модуля subprocess, который позволил не только реализовать вызовы команд, но и естественным образом перенаправить их вывод в указанные файлы.

При реализации поддержки проверки условий помимо элементарных операций, таких как поиск подстрок в файле, проверка существования файла и сравнения файлов, средствами модуля numpy был реализован разбор файлов данных в формате Tecplot с последующим анализом полей переменных.

Поддержка рассылки оповещений была реализована с использованием средств модулей smtplib и email.

Работа с SVN-хранилищем реализована в системе посредством вызовов команд svn средствами модуля subprocess.

Помимо проверки условий успешного завершения отдельных тестов, в системе реализованы проверка успешной компиляции исходных кодов модулей, библиотек и утилит входящих в состав ПК NOISEtte.

Программный код самой системы тестирования так же, как код и тесты, хранится в SVN-хранилище. Однако при этом в целях безопасности и удобства настройки и конфиденциальные данные, такие как пути и параметры доступа к SVN, почтовые адреса разработчиков и пути к рабочим каталогам, вынесены в отдельный конфигурационный файл вне SVN.

Непосредственно процесс запуска тестов в системе, при условии, что код ПК успешно скомпилирован и тесты успешно извлечены их SVN-хранилища во временный локальный каталог (далее – «рабочий каталог»), происходит следующим образом (см. Листинг 2): программа обходит все дерево каталогов под рабочим каталогом посредством метода системного модуля os.walk(), и каждый каталог, в котором присутствует файл “testcase.xml”, трактуется как каталог, содержащий очередной тест (строки 1, 2). Производится разбор конфигурационного файла “testcase.xml”, инициализируются экземпляры классов QATestCase, и производится попытка запуска теста (строки 14 – 16 листинга 2). Результаты теста копируются в заранее определенный каталог результатов, а краткий отчет о выполнении теста записывается в единый журнал (строки 27 – 29), и происходит переход к обработке следующего теста.

Краткий файл отчета о тестировании, заполненный в ходе выполнения те-

```
for root, dirnames, filenames in os.walk(currqa_path):
  for filename in fnmatch.filter(filenames, 'testcase.xml'):
    # testcase with xml found
    testcasepath=pjoin(root, filename)
    h=root
    dirs=[]
    while currqa_dir in h:
      h,f = os.path.split(h)
      if not currqa_dir in f: dirs.append(f)
    results_path=qarunlogs_path
    dirs.reverse()
    for d in dirs: results_path=pjoin(results_path,d)

    test=QATestCase(testcasepath,os.path.abspath(sandbox_path))
    results={}
    teststatus=test.run(results_path,results)
    total_cases+=1
    if teststatus: passed_cases+=1

    for runname in results.keys():
      res_html=""
      status=""
      mode=results[runname]["mode"]
      name=results[runname]["name"]
      status=results[runname]["status"]

      for s in results[runname].keys():
        if s=="results":
          for ss in results[runname][s]: res_html+=ss.replace(
             qarunlogs_path,"<QABASEDIR>")+"|"

      log_qa_status(name,status,res_html+mode,not teststatus)
if (total_cases>passed_cases):
  msg_tests_stat=str(total_cases-passed_cases)+" of "+str(
     total_cases)+" cases FAILED"
else:
  msg_tests_stat=str(total_cases)+" cases PASSED"
msgtext=msgtext+"\n"+msg_tests_stat+"\n"
```

Листинг 2. Фрагмент кода, отвечающего за выполнение тестов

```
1  STARTED|AT|2018-02-20 23:00:01
2  NEW_REVISION|PASSED|4185(2018-02-20 19:18:14 +0300 by cherepock)
3  CODE_CHECKOUT|PASSED|<QABASEDIR>/code_checkout.log
4  QA_CHECKOUT|PASSED|<QABASEDIR>/qa_checkout.log
5  COMPILATION|PASSED|<QABASEDIR>/compilation.log
6  QA_STIEFEL 3D case012|PASSED|<QABASEDIR>/QA_STIEFEL_3D/case012/4
      xMPI/QA_log.txt|<QABASEDIR>/QA_STIEFEL_3D/case012/4xMPI/testcase
      .stdout|4xMPI
7  QA_STIEFEL 3D case012|PASSED|<QABASEDIR>/QA_STIEFEL_3D/case012/SP/
      QA_log.txt|<QABASEDIR>/QA_STIEFEL_3D/case012/SP/testcase.stdout|
      SP
8  QA_UTILS linkmesh|FAILED|<QABASEDIR>/QA_utils/linkmesh/SP/checkmesh
      .std|<QABASEDIR>/QA_utils/linkmesh/SP/testcase.stdout|<QABASEDIR
      >/QA_utils/linkmesh/SP/util.std|SP
9  QA_UTILS approx|PASSED|<QABASEDIR>/QA_utils/approx/SP/approximation
      .std|<QABASEDIR>/QA_utils/approx/SP/meshperiodic.std|<QABASEDIR
      >/QA_utils/approx/SP/refinemesh.std|<QABASEDIR>/QA_utils/approx/
      SP/util_genmesh.std|SP
10 QA_UTILS interpolation.per2noper|FAILED|<QABASEDIR>/QA_utils/
      interpolation.per2noper/SP/checkmesh.std|<QABASEDIR>/QA_utils/
      interpolation.per2noper/SP/testcase.stdout|<QABASEDIR>/QA_utils/
      interpolation.per2noper/SP/util.std|SP
```

Листинг 3. Фрагмент отчета тестирования

стирования, представлен в листинге 3.

В заголовке указаны общие данные: время начала тестирования (строка 1 листинга 3), результат извлечения текущий версии кода ПК из SVN и разработчик, вносивший последние изменения в код (строки 2, 3), результат извлечения текущий версии набора тестов из SVN (строка 4) и результат компиляции кода (строка 5). Далее по одному в каждой строке следуют отчеты о выполнении каждого сценария для каждого теста, в которых указаны название теста, статус выполнения (“PASSED” – завершился успешно, “FAILED” – завершился неуспешно), список файлов-результатов теста, и в конце следует пометка о режиме выполнения теста (“SP” – однопроцессорный режим, “4xMPI” – многопроцессорный).

Система тестирования была установлена и настроена на выделенном сервере на платформе Linux с операционной системой Ubuntu 14.04 LTS. Были установлены и настроены SVN-клиент для доступа к SVN-хранилищу, Python 3.4.3 с необходимыми модулями, а также настроен почтовый сервер. С использованием системы cron был настроен ежедневный запуск системы тестирования. Заметим, что при такой организации хранения тестов и процесса тестирования добавление разработчиком нового теста заключается в подготовке сценария и

внесении его в общий набор вместе с конфигурационным файлом. При следующем запуске тестирования новый тест будет автоматически включен в тестовый набор.

Система автоматического тестирования ПК NOISEtte: веб-интерфейс

Для более удобного представления результатов тестирования был реализован доступ к результатам тестирования через веб-интерфейс. На сервере, на котором была установлена и работала система тестирования, был настроен веб-сервер Apache 2.4. С использованием средств `JavaScript` и библиотеки `jQuery` был разработан веб-интерфейс для отображения результатов работы системы тестирования.

В качестве связующего звена между веб-интерфейсом и результатами тестов была написана вспомогательная программа на языке `Python`, которая принимает один параметр. В зависимости от значения этого параметра, программа возвращает в формате `JSON` либо список ревизий, для которых есть результаты тестов, либо результаты тестирования для указанной в параметре версии кода.

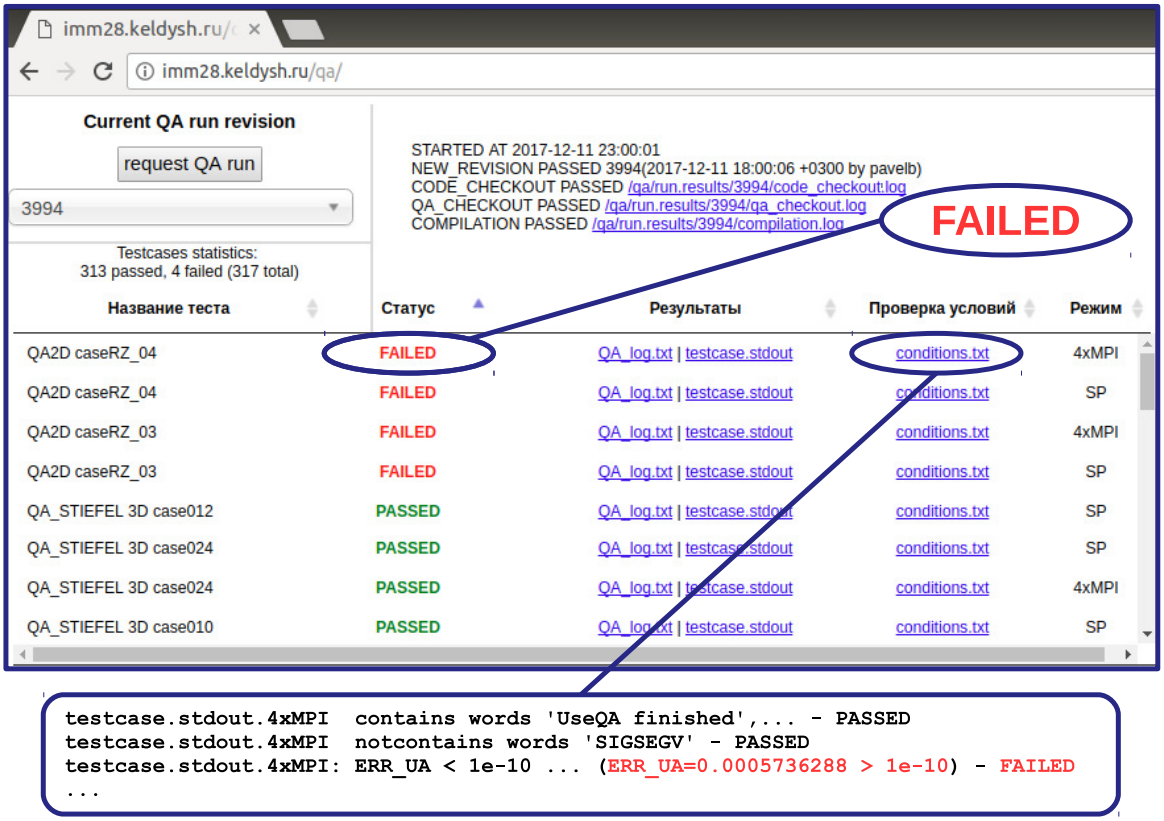

Рис. 3. Общий вид web-интерфейса системы тестиования ПК NOISEtte

При загрузке страницы через `GET`-запрос сначала вызывается вспомогатель-

ная программа для заполнения и отображения списка версий кода, для которых доступны результаты тестирования, затем, также посредством GET-запроса к той же программе, извлекаются результаты тестирования для наиболее поздней версии кода. Общий вид интерфейса представлен на рис. 3. Вся страница представляет собой таблицу, в которой при загрузке отображаются результаты тестирования самой поздней из доступных версий кода. При желании пользователь может выбрать и посмотреть результаты тестирования для более ранних версий кода, используя выпадающий слева-сверху список доступных ревизий кода. В таблице отображаются все тесты, выполненные для выбранной ревизии кода, их названия, статусы завершения, ссылки на файлы результатов и файлы анализа выполнения условий. На рис. 3 показано, что статусы некорректно завершившихся тестов выделены красным цветом, и показано, что по содержанию файла “conditions.txt”, доступного по ссылке через веб-интерфейс, можно понять, какие именно условия были нарушены.

Также для удобства разработчиков в интерфейсе реализована возможность запроса запуска внеочередного раунда тестирования, что может быть полезным, если разработчик внес изменения в код и желает немедленно проверить результаты тестирования нового кода.

Заключение

В результате работы над улучшением и автоматизацией процесса регрессионного тестирования кода ПК NOISEtte была создана кроссплатформенная система автоматического регрессионного тестирования.

Разработанная система тестирования позволяет как в автоматическом режиме постоянно проводить тестирование кода и извещать разработчиков в случае деградации кода, так и проводить локальное тестирование в случае необходимости.

При этом система обладает информативным веб-интерфейсом, способным как отображать текущие результаты тестирования, так и анализировать всю историю результатов для предыдущих версий кода.

Разработанная система применима для проведения кроссплатформенного регрессионного тестирования произвольных программных продуктов.

Список литературы

1. *The CVS Team* Concurrent Versions System (CVS). — 1990-2008. — URL: http://www.nongnu.org/cvs.
2. *Apache Software Foundation* Subversion(SVN). — 2004. — URL: https://subversion.apache.org.
3. *Mackall M.* Mercurial. — 2005. — URL: https://www.mercurial-scm.org.
4. *Torvalds L., Hamano J.* Git. — 2005. — URL: https://git-scm.com.
5. Параллельный программный комплекс NOISETTE для крупномасштабных расчетов задач аэродинамики и аэроакустики / И. В. Абалакин, П. А. Бахвалов, А. В. Горобец, А. П. Дубень, Т. К. Козубская // Вычислительные методы и программирование. — 2012. — Т. 13, № 2. — С. 110—125.
6. *Lang J.-P.* Redmine. — 2006. — URL: https://www.redmine.org.
7. *Edgewall Software* Trac. — 2004. — URL: https://trac.edgewall.org.
8. *Atlassian* JIRA. — 2002. — URL: https://www.atlassian.com/software/jira.
9. *37signals* Basecamp. — 2004. — URL: https://basecamp.com.
10. *Guido van Rossum, Python Software Foundation* Python. — 1991. — URL: https://www.python.org.
11. *World Wide Web Consortium* XML(eXtensible Markup Language). — 1998. — URL: https://www.w3.org/XML.