Валерий Аджиев

Мифы о безопасном ПО: уроки знаменитых катастроф

«Если бы строители строили
здания так же, как
программисты пишут
программы, первый залетевший
дятел разрушил бы
цивилизацию»

Второй закон Вейлера

Не секрет, что ошибки в программном обеспечении «ответственных» систем могут вызвать чрезвычайные последствия, тем не менее, в обществе, особенно на уровне массового потребителя ИТ, продолжает витать иллюзия непогрешимости компьютера и работающего на нем ПО. В статье подробно разбираются две вошедших в историю компьютерной индустрии катастрофы и обсуждаются некоторые мифы, связанные с такими понятиями, как безопасность и риски в контексте разработки и эксплуатации программно-аппаратных систем.

Слово «безопасность» не сходит со страниц компьютерной прессы. Однако, употребляется оно обычно в контексте поддержания целостности данных и особенно обеспечения их конфиденциальности. Что ж, тема интересная: Internet, банки, спецслужбы, хакеры… все, к чему приложимо вошедшее в повседневный обиход слово «security». Есть, однако, у «безопасности» и другое измерение, чаще обозначаемое не столь популярным термином «safety», про которое говорят меньше, но важность его применительно к компьютерным системам поистине нельзя переоценить.

В мире постоянно происходят катастрофы, большие, малые аварии и все чаще их причиной становится ненадлежащее функционирование компьютерных систем и, в частности, их программного обеспечения. Оборона,

авиация и космос, медицина, технологические процессы на современных ядерных, химических и других производствах вот неполный перечень тех предметных областей, где низкое качество ПО и даже единичные дефекты в нем находят воплощение в терминах потерянных человеческих жизней и разрушенных материальных ценностей.

Над такого рода «ответственными» (mission-critical) системами работает целая отрасль, в которой крутятся очень большие (по-преимуществу, бюджетные) деньги и где как справедливо принято считать сосредоточено значительное количество высококвалифицированных программистов и проектировщиков, надлежащим образом поставлен менеджмент, отлажены процессы разработки и контроля. И тем не менее, «кое-где... порой...» ПО дает сбой, и резонанс тогда бывет громкий. Разберем две знаменитых истории, в одной из которых программистские ошибки привели к беспрецедентным материальным потерям, в другой к смерти нескольких человек, и попытаемся за этими частными случаями увидеть некоторые общие проблемы, стоящие сегодня перед всей программной индустрией.

Катастрофа Ariane 5

4 июня 1996 г. был произведен первый запуск ракеты-носителя Ariane 5 детища и гордости Европейского Сообщества. Уже через неполные 40 сек. все закончилось взрывом. Автоподрыв 50-метровой ракеты произошел в районе ее запуска с космодрома во Французской Гвиане. За предшествующие годы ракеты серии Ariane семь раз терпели аварии, но эта побила все рекорды по вызванным ею убыткам. Только находившееся на борту научное оборудование потянуло на пол-миллиарда долларов, не говоря о прочих разноообразных издержках; а астрономические цифры «упущенной выгоды» от несостоявшихся коммерческих запусков и потеря репутации надежного перевозчика в очень конкурентном секторе мировой экономики («стоимость рынка» к 2000 г. должна превысить 60 млрд. долл.) с трудом поддаются оценке. ебезынтересно отметить, что предыдущая модель ракета Ariane 4 успешно запускалась более 100 раз.

Буквально на следующий день Генеральный директор Европейского Космического Агенства (ESA) и Председатель Правления Французского ационального Центра по изучению Космоса (CNES) издали распоряжение об образовании независимой Комиссии по Расследованию обстоятельств и причин этого чрезвычайного происшествия, в которую вошли известные специалисты и ученые изо всех заинтересованных европейских стран. Возглавил Комиссию представитель Французской Академии аук профессор Жак-Луи Лион (Jacques-Louis Lions).

Кроме того, был сформирован специальный Технический Комитет из представителей заказчиков и подрядчиков, ответственных за производство и эксплуатацию ракеты, в чью обязанность было вменено незамедлительно предоставлять Комиссии всю необходимую информацию.

13 июня 1996 г. Комиссия приступила к работе, а уже 19 июля был обнародован ее исчерпывающий доклад, который сразу же стал доступен в Сети.[1] Что же касается информации, которую при участии нескольких институтов осмысляла Комиссия, то она состояла из телеметрии, траекторных данных, а также оптических наблюдений за ходом полета. Были собраны (что само по себе было непросто, так как взрыв произошел на высоте приблизительно 4 км, и осколки были рассеяны на площади около 12 кв. км. в саванне и болотах) и изучены части ракеты и оборудования. Кроме того, были заслушаны показания многочисленных специалистов и изучены горы производственной и эксплуатационной документации.

Технические подробности аварии

Положение и ориентация ракеты-носителя в пространстве измеряются авигационной Системой (Inertial Reference Systems IRS), составной частью которой является встроенный компьютер, вычисляющий углы и скорости на основе информации от бортовой Инерциальной Платформы, оборудованной лазерными

[1] «Ariane 5: Flight 501 Failure», http://www.eIRSn.esa.it/htdocs/tidc/press/Press96/press33.html

гироскопами и акселерометрами. Данные от IRS передаются по специальной шине на Бортовой Компьютер (On-Board Computer OBC), который обеспечивает необходимую для реализации программы полета информацию и непосредственно через гидравлические и сервоприводы управляет твердотопливными ускорителями и криогенным двигателем типа Вулкан (Vulkain).

Как обычно, для обеспечения надежности Системы Управления Полетом используется дублирование оборудования. Поэтому две системы IRS (одна активная, другая ее горячий резерв) с идентичным аппаратным и программным обеспечением функционируют параллельно. Как только бортовой компьютер OBC обнаруживает, что «активная» IRS вышла из штатного режима, он сразу же переключается на другую. Впрочем, и бортовых компьютеров тоже два.

Теперь, следуя Докладу Комиссии,[2] проследим все значимые фазы развития процесса, оказавшегося в конце концов аварийным. Момент старта обозначим H0 — это и будет точка отсчета для всех событий, хотя отслеживать их мы будем в обратном начиная с момента саморазрушения системы порядке. Для полноты картины упомянем, что предшествующие старту операции происходили в нормальном режиме вплоть до момента H0-7 минут, когда было зафиксировано нарушение «критерия видимости». Поэтому старт был перенесен на час; в H0 = 9 час. 33 мин. 59 сек. местного времени «окно запуска» было вновь «поймано» и был, наконец, осуществлен сам запуск, который и происходил штатно

[2] См. сноску 1

вплоть до момента H0+37 сек. В последующие секунды произошло резкое отклонение ракеты от заданной траектории, что и закончилось взрывом. Итак:

* в момент H0+39 сек. из-за высокой аэродинамической нагрузки вследствие превышения «углом атаки» критической величины на 20 градусов произошло отделение стартовых ускорителей ракеты от основной ее ступени, что и послужило основанием для включения Системы Автоподрыва ракеты;

* изменение угла атаки произошло по причине нештатного вращения сопел твердотопливных ускорителей;

* такое отклонение сопел ускорителей от правильной ориентации вызвала в момент H0 + 37 сек. команда, выданная Бортовым Компьютером на основе информации от активной авигационной Системы (IRS 2). Часть этой информации была в принципе некорректной: то, что интерпретировалось как полетные данные, на самом деле являлось диагностической информацией встроенного компьютера системы IRS 2;

* встроенный компьютер IRS 2 передал некорректные данные, потому что диагностировал нештатную ситуацию, «поймав» исключение (exception), выброшенное одним из модулей программного обеспечения;

* при этом Бортовой Компьютер не мог переключиться на резервную систему IRS 1, так как

она уже прекратила функционировать в течение предшествующего цикла (занявшего 72 мсек.) по той же причине, что и IRS 2;

* исключение, «выброшенное» одной из программ IRS, явилось следствием выполнения преобразования данных из 64-разрядного формата с плавающей точкой в 16-разрядное целое со знаком, что привело к «Operand Error»;

* ошибка произошла в компоненте ПО, предназначенном исключительно для выполнения «регулировки» Инерциальной Платформы. Причем что звучит парадоксально, если не абсурдно этот программный модуль выдает значимые результаты только до момента H0 + 7 сек. отрыва ракеты со стартовой площадки. После того, как ракета взлетела, никакого влияния на полет функционирование данного модуля оказать не могло;

* однако, «функция регулировки» действительно должна была (в соответствии с установленными для нее требованиями) действовать еще 50 сек. после инициации «полетного режима» на шине авигационной Системы (момент H0-3 сек.), что она с усердием дурака, которого заставили богу молиться, и делала;

* ошибка «Operand Error» произошла из-за неожиданно большой величины BH (Horizontal Bias горизонтальный наклон), посчитанной внутренней функцией на основании величины «горизонтальной

скорости», измеренной находящимися на Платформе датчиками. Величина BH служила индикатором точности позиционирования Платформы;

* величина BH оказалась много больше, чем ожидалось потому, что траектория полета Ariane 5 на ранней стадии существенно отличалась от траектории полета Ariane 4 (где этот программный модуль использовался ранее), что и привело к значительно более высокой «горизонтальной скорости».

Финальным же действием, имевшим фатальные последствия, стало прекращение работы процессора; соответственно, вся авигационная Система перестала функционировать. Возобновить же ее действия оказалось технически невозможно.

Осталось добавить, что всю эту цепь событий удалось полностью воспроизвести с помощью компьютерного моделирования, что вкупе с материалами других исследований и экспериментов позволило заключить; причины и обстоятельства катастрофы полностью выявлены.

Причины и истоки аварии

Прежде всего проследим, откуда взялось первоначальное требование на продолжение выполнения операции регулировки после взлета ракеты.

Оказывается, оно было заложено более чем за 10 лет до рокового события, когда проектировались еще ранние модели серии Ariane. При некотором (весьма

маловероятном!) развитии событий взлет мог быть отменен буквально за несколько секунд до старта, например в промежутке H0-9 сек., когда на IRS запускался «полетный режим», и H0-5 сек., когда выдавалась команда на выполнение некоторых операций с ракетным оборудованием. В случае неожиданной отмены взлета необходимо было быстро вернуться в режим «обратного отсчета» (countdown) и при этом не повторять сначала все установочные операции, в том числе приведение к исходному положения Инерциальной Платформы (операция, требующая 45 мин. время, за которое можно потерять «окно запуска»).

Было обосновано, что в случае события отмены старта период в 50 сек. после H0-9 будет достаточным для того, чтобы наземное оборудование смогло восстановить полный контроль за Инерциальной Платформой без потери информации за это время Платформа прекратит начавшееся было перемещение, а соответствующий программный модуль всю информацию о ее состоянии зафиксирует, что поможет оперативно возвратить ее в исходное положение (напомним, что все это в случае, когда ракета продолжает находиться на месте старта). И действительно, однажды, в 1989 г., при старте под номером 33 ракеты Ariane 4, эта особенность была с успехом задействована.

Однако, Ariane 5, в отличие от предыдущей модели, имел уже принципиально другую дисциплину выполнения предполетных действий настолько другую, что работа рокового программного модуля после времени старта вообще не имела смысла. Однако, модуль повторно использовался без каких-либо модификаций видимо из-за

нежелания изменять программный код, который успешно работает.

В конце концов, было бы странно, если бы тривиальная ошибка переполнения (даже если она и возникла) была бы столь фатальной, что с ней невозможно бороться. Почему же программный код (написанный на таком оснащенном всеми необходимыми для обеспечения надежности средствами языке, как Ада) оказался незащищеным до такой степени, что наступили столь катастрофические последствия?

Расследование показало, что в данном программном модуле присутствовало целых семь переменных, вовлеченных в операции преобразования типов. Оказалось, что разработчики проводили анализ всех операций, способных потенциально генерировать исключение, на уязвимость. И это было их вполне сознательным решением добавить надлежащую защиту к четырем переменным, а три включая BH, оставить незащищенными. Основанием для такого решения была уверенность в том, что для этих трех переменных возникновение ситуации переполнения невозможно в принципе. Уверенность эта была подкреплена расчетами, показывающими, что ожидаемый диапазон физических полетных параметров, на основании которых определяются величины упомянутых переменных, таков, что к нежелательной ситуации привести не может. И это было верно но для траектории, рассчитанной для модели Ariane 4. А ракета нового поколения Ariane 5 стартовала по совсем другой траектории, для которой никаких оценок не выполнялось. Между тем она (вкупе с высоким начальным ускорением) была такова, что

«горизонтальная скорость» превзошла расчетную (для Ariane 4) более чем в пять раз.

Но почему же не была (пусть в порядке перестраховки) обеспечена защита для всех семи, включая BH, переменных? Оказывается, для компьютера IRS была продекларирована максимальная величина рабочей нагрузки в 80 %, и поэтому разработчики должны были искать пути снижения излишних вычислительных издержек. Вот они и ослабили защиту там, где теоретически нежелательной ситуации возникнуть не могло. Когда же она возникла, то вступил в действие такой механизм обработки исключительной ситуации, который оказался совершенно неадекватным.

Этот механизм предусматривал следующие три основных действия. Прежде всего, информация о возникновении нештатной ситуации должна быть передана по шине на бортовой компьютер OBC; параллельно она вместе со всем контекстом записывалась в перепрограммируемую память EEPROM (которую во время расследования удалось восстановить и прочесть ее содержимое), и наконец, работа процессора IRS должна была аварийно завершиться. Последнее действие и оказалось фатальным именно оно, случившееся в ситуации, которая на самом деле была нормальной (несмотря на сгенерированное из-за незащищенного переполнения программное исключение), и привело к катастрофе.

Осмысление

Произошедшая с Ariane 5 катастрофа имела исключительно большой резонанс и по причине беспрецедентных материальных потерь, и вследствие

очень оперативного расследования, характеризовавшегося к тому же открытостью результатов (впервые такая практика публичности была опробована при расследовании причин аварии космического корабля Challenger 1986 г.). Сразу стало очевидным, что данному событию суждено войти в историю не только космонавтики, но и программной инженерии. Поэтому неудивительно, что авария послужила поводом для оживленной дискуссии, в которой приняли участие многие известные специалисты.

Ж.-М. Жезекель (J.-M. Jezequel) и Б. Мейер (B.Meyer)[3] пришли к совершенно однозначному выводу: допущенная (и так и не выявленная) программная ошибка носит, по их мнению, чисто технический характер и коренится в некорректной практике повторного использования ПО. Более точная формулировка: роковую роль сыграло отсутствие точной спецификации повторно-используемого модуля.

Расследование показало, что обнаружить требование, устанавливающее максимальную величину ВН (вмещающуюся в 16 битов), можно было с большим трудом: оно затерялось в приложениях к основному спецификационному документу. Мало того, в самом коде на этот счет не было никаких комментариев, не говоря уже о ссылке на документ с обоснованием этого требования.

В качестве панацеи в такого рода ситуациях авторы предложили задействовать принцип «Контрактного Проектирования» (что и неудивительно, ибо его

[3] J.-M. Jezequel, B. Meyer «Put It in the Contract: The Lessons of Ariane», // Computer, Vol.30, No.2, January 1997, pp.129–130

разработчиком как раз и является Мейер[4]). Именно «контракт» в духе языка Eiffel, явным образом (с помощью пред- и пост-условий) устанавливающий для любого программного компонента ограничения на входные и выходные параметры, и мог бы предотратить катастрофическое развитие событий. Был приведен и набросок такого контракта:

```
convert (horizontal_bias: INTEGER): INTEGER is
    require
        horizontal_bias <= Maximum_bias
            do
            ...
        ensure
    ...
end
```

Соответственно, ошибка могла быть выявлена уже на этапе тестирования и отладки (когда проверка логических утверждений включается по специальной опции компилятора); если же пред- и пост-условия проверялись бы и во время полета, то сгенерированное исключение могло быть надлежащим образом обработано (правда, авторы оговариваются, что использование такого режима могло бы нарушить ограничения, связанные с вычислительной нагрузкой).

[4] Б. Мейер «Построение надежного объектно-ориентированного ПО. Введение в Контрактное Проектирование», //Открытые Системы, № 6, 1998

Однако, самым важным достоинством использования контрактных механизмов является, по мнению авторов, явное присутствие легко понимаемых и при необходимости верифицируемых ограничений как в документации, так и в коде.

При работе над сложными проектами типа Ariane именно контракты могли бы выступать в качестве опорных ориентиров для групп качества «QA Team», чья задача выполнять систематический мониторинг ПО на предмет соответствия требованиям. Авторы с сожалением заключают, что контрактные механизмы никак не получат должного распространения в современной практике. Более того, положение только усугубляется: например, в Java даже исчезла присутствовавшая в языке Си скромная по возможностям инструкция «assert». В составной части CORBA языке IDL (Interface Definition Language), предназначенном обеспечить полномасштабное повторное использование компонентов в распределенной среде, отсутствует какой-либо механизм спецификации семантики. То же относится и к ActiveX. Авторы заключают: без полной и точной спецификации, основанной на пред- и пост-условиях и инвариантах, «повторное использование программных компонентов совершенное безрассудство».

Эта точка зрения вызвала многочисленные отклики. Хотя полезность использования контрактных механизмов никто не оспаривал, все же взгляд авторов многим показался упрощенным. Наиболее обстоятельный критический разбор их статьи выполнил сотрудник Locheed Martin Tactical AirCraft Systems, известный специалист в области разработки ответственных систем

Кен Гарлингтон (Ken Garlington).[5] Он начал с того, что указал на ошибку в приведенном наброске контракта, где предполагается, что BH преобразуется не из вещественного (как то было в реальности) числа, а из целого.

Показательно, пишет Гарлингтон, что он оказался первым, кто обратил внимание на столь очевидный прокол, а ведь статью читали и публично обсуждали многие квалифицированные специалисты. С тем же успехом (а точнее неуспехом) могла пройти мимо этого дефекта и «QA-team». Так что даже точная спецификация сама по себе не панацея. Гарлингтон также подробно разобрал нетривиальные проблемы, возникающие при написании не «наброска», а действительно полной спецификации контракта для данной конкретной ситуации.

Вывод Гарлингтона вполне отвечает здравому смыслу: проблема носит комплексный характер и обусловлена прежде всего объективной сложностью систем типа Ariane. Соответственно, одним лекарством болезнь, приводящая к появлению ошибок в ПО, вылечена быть не может. Хотя то, что процесс мониторинга спецификаций, кода и документов с обоснованием проектных решений при разработке ПО для Ariane 5, оказался неадекватен, отметила и Комиссия по расследованию аварии. В частности, подчеркнуто, что к процессу контроля не привлекались специалисты из организаций, независимых как от заказчика, так и от

[5] K. Garlington «Critique of „Put it in the Contract: The Lessons of Ariane"», March 1998 http://www.flash.net/~kennieg/ariane.html

подрядчика системы, что нарушило принцип разделения исполнительных и контрольных функций.

Большие претензии были предъявлены не только к процессу тестирования как таковому, но и к самой его стратегии. На этапе тестирования и отладки системы было технически возможно в рамках интегрального моделирования процесса полета исследовать все аспекты работы IRS, что позволило бы почти гарантированно выявить ошибку, приведшую к аварии. Однако, вместо этого при моделировании работы всего комплекса IRS рассматривалась как черный ящик, заведомо выдающий то, что ожидается. Почему? А зачем тестировать то, что успешно работало в течение многих лет?!

Было обращено внимание и на невыявленную при анализе требований к проекту взаимную противоречивость между необходимостью обеспечения надежности и декларацией о величине максимально допустимой нагрузки на компьютер, что и явилось предпосылкой принятия программистами потенциально опасного компромиссного решения о защите от переполнения не всех семи, а только четырех переменных. Впрочем, как справедливо замечает Б. Мейер, всякий инженерный процесс предполагает принятие компромиссных решений в условиях множества разноречивых требований; вопрос в том, насколько полна информация, на основании которой такие решения принимаются.

Особый разговор о механизме обработки исключительных ситуаций, который, как уже говорилось, жил своей особой жизнью в отрыве от общего контекста всей ситуации с полетом, и в итоге уподобился тому врачу, что без всякого осмотра пристрелил пришедшего к

нему с непонятными симптомами больного, дабы тот не мучился. Реализация именно такого механизма явилась следствием распространенной при разработке «ответственных» систем проектной культуры особо и радикально реагировать на возникновение случайных аппаратных сбоев.

Принцип действий здесь «простой, как мычание», исходящий из критериев безусловного обеспечения максимальной надежности: отключать допустившее сбой оборудование и тут же задействовать резервный блок: вероятность того, что этот блок также выдаст случайный сбой, да еще в той же ситуации, для надлежаще спроектированных и отлаженных аппаратных систем чрезвычайна мала.

В данном же случае, возникла систематическая программная ошибка; «систематическая» в том смысле, что при повторении тех же входных условий, она обязательно возникнет вновь, ибо тавтология здесь уместна запрограммирована. Вот почему подключение резервной авигационной Системы ничего не дало: ведь ПО на нем исполнялось то же самое, соответственно и обе IRS, и дублирующие друг друга Бортовые Компьютеры с неотвратимостью натыкались на ту же ситуацию и с покорностью обреченных на заклание овец шли к катастрофе. Очевидно, что необходимо по крайней мере отнять у «врача» пистолет: прекращать функционирование аппаратуры при возникновении программного «исключения» целесообразно лишь после комплексного анализа ситуации, но никак не автоматически.

В контексте данной статьи интересно мнение главного редактора журнала «Automated Software

Engineering» Башара узейбеха (Bashar Nuseibeh),[6] который, дав обзор разных точек зрения на причины аварии и придя к выводу, что «все правы», связал все-таки катастрофу Ariane 5 с более общими проблемами разработки программных систем. Он отметил, в частности, что современные тенденции в программной инженерии, связанные с разделением интересов вовлеченных в разработку независимо работающих персонажей (что связано с широким внедрением таких подходов, как объектно-ориентированные и компонентные технологии) не получают надлежащего балансирующего противовеса в виде менеджмента, способного координировать всю работу на должном уровне.

Эта тема заслуживает дальнейшего обсуждения, но сначала обратимся к еще одной печально знаменитой истории.

---

[6] B. Nuseibeh «Ariane 5: Who Dunnit?», //IEEE Software, Vol.14, No.3, 1997, pp.15–16

Инциденты с Therac-25

Вспомним более давнюю историю, почти во всем отличную от ситуации с Ariane 5, а сходную только в том, что она также была подробно задокументирована[7] и получила в свое время большой резонанс как повлекшая наиболее тяжкие последствия за всю не столь долгую историю использования медицинских комплексов, управляемых компьютерами. Правда в этом случае полномасштабного официального расследования проведено не было; источниками информации послужили, в основном, протоколы встреч пользователей системы с производителем и материалы многочисленных судебных разбирательств.

Технические подробности инцидентов

В 1985-87 гг. 6 человек получили смертельную дозу облучения во время сеансов радиационной терапии с применением медицинского ускорителя Therac-25 (количество пациентов, также подвергшихся переоблучению, но выживших, точно не известно). Производителем установки являлось одно из подразделений Канадского Агентства Атомной Энергии (Atomic Energy of Canada Limited AECL).

Модель Therac-25, законченная в виде прототипа в 1976 г. и поступившая в промышленную эксплуатацию в 1982 г. (пять установок в США и шесть в Канаде) была

[7] N. Leveson, C. Turner «An Investigation of the Therac-25 Accidents», — Computer, Vol.26, N.7, July 1993, p. 18–41

развитием ранних моделей Therac-6 и Therac-20. При этом некоторые программные модули, разработанные для ранних моделей, использовались повторно (в том числе поставленные партнером, французской фирмой CGR, сотрудничество AECL с которой прекратилось к моменту начала работ над Therac-25).

Первый зафиксированный факт переоблучения, случившийся в Онкологическом Центре в Marietta (штат Джорджия) в июне 1985 г., просто отрицался и не был должным образом исследован: производитель с цифрами оценки рисков в руках утверждал, что на данной системе это просто невозможно. По странному совпадению, проведенный сеанс облучения не был документирован, так как почему-то вышел из строя принтер; в результате поданный родственниками пациента иск не получил хода ввиду отсутствия документальных доказательств, хотя доза облучения по оценкам была превышена в 100 раз.

Следующий инцидент, случившийся в июле того же года в Онкологическом Центре Онтарио как раз был задокументирован хорошо, но производитель не смог воспроизвести ситуацию, и ее отнесли на счет случайного сбоя аппаратуры; в ПО сомнений по-прежнему просто не было. И трагические инциденты продолжились.

Очередной из них произошел в Онкологическом Центре Восточного Техаса в марте 1986 г. В данном случае процессом управляла опытный оператор, проведшая уже более 500 подобных сеансов. Она быстро ввела предписанные параметры, после чего заметила, что вместо режима облучения электронными лучами заказала лучи рентгеновские (которыми пользовали большинство пациентов). Коррекция требовала

исправления всего одной буквы; нажав кнопку, она вошла в режим редактирования, скорректировала в нужном месте «x» на «e», затем несколькими нажатиями клавиши «Return» (благо, все остальные параметры были введены правильно) достигла нижней (командной) строки экрана, убедилась, что против каждого введенного параметра горит «VERIFIED», а статус системы ожидаемый («BEAM READY»), и выдала команду начать процесс облучения. Однако, неожиданно система встала, на консоли высветилось сообщение «MALFUNCTION 54», а статус системы изменился на «TREATMENT PAUSE», что свидетельствовало о проблеме невысокой степени серьезности. Висевшая тут же бумага с кодами ошибок «исчерпывающе» поясняла, что «MALFUNCTION 54» означает «dose input 2». Забегая вперед, укажем, что много позже, во внутренней документации производителя было обнаружено, что это сообщение выдавалось в случае «ненадлежащей дозы облучения» причем, как для слишком большой, так и для слишком малой, что само по себе странно (да и просто недопустимо ведь ситуации принципиально разные).

Озадаченная операторша взглянула на высветившееся количество отпущенной дозы и увидела, что оно пренебрежимо мало. Поэтому она без долгих раздумий выдала команду на продолжение процесса, после чего вся описанная выше ситуация повторилась.

Тем временем пациент, который возлежал на столе в изолированном от оператора помещении, испытал некое подобие электрического шока. Он тоже был опытным (для него это был девятый сеанс), поэтому понял, что творится что-то неладное. Однако, дать сразу же знать об этом оператору через специально для того предназначенные видео и аудио средства он не смог: как

выяснилось, видео было по непонятным причинам отключено, а аудиоканал просто неисправен.

После повторного шокового удара пациент вскочил и нимало шокировал уже операторшу, начав ломиться в стеклянные двери ее помещения. Поначалу его и лечили от электрошока (он умер через пять месяцев). Позднейшее моделирование ситуации показало, что пациент получил менее чем за 1 сек. на участок позвоночника в 1 кв. см. дозу в диапазоне от 16500 до 25000 рад (в то время, как ему было предписано принять в этом сеансе 180 рад, а всего 6000 рад за шесть с половиной недель).

Прибывший из AECL инженер, несмотря на все усилия, оказался не в состоянии воспроизвести ситуацию, хотя заверил, что переоблучение в принципе невозможно. Были успешно прогнаны все тесты, система снова вступила в эксплуатацию, и через три недели инцидент повторился во всех деталях с тем же трагическим результатом. Только после этого установка была выведена из эксплуатации, и началось углубленное расследование, шедшее, кстати, очень трудно. Опуская множество деталей, приведем его итоги, интересные с программистской точки зрения.

Особенности ПО как предпосылки для инцидентов

В комплексе не использовалась какая-либо стандартная операционная система: была разработана специальная мультизадачная ОС реального времени, для компьютера PDP-11/23 с 32Кбайт и написанная на языке ассемблера. Специальный планировщик координировал

деятельность всех одновременно исполняющихся процессов. Задачи, запускавшиеся каждые 0.1 сек., разделялись на «критические», исполнявшиеся первыми, и «некритические». К критическим отнесены три приоритетных задачи (рис. 1):

* **«Servo»**, ответственная за все операции, связанные с эмиссией радиационных пучков и доставкой их к месту назначения;

* **«Housekeeper»**, выполнявшая верификацию всех параметров и ответственная за блокировку работы в случае возникновения нештатной ситуации, а также за сообщения о таких ситуациях;

* **«Treat»**, управлявшая самим процессом лечения, который был разделен на 8 операционных фаз. В зависимости от значения переменной Tphase вызывалась одна из восьми подпрограмм, по окончании работы которой Treat в зависимости от значений нескольких разделяемых с другими критическими и некритическими задачами переменных, вырабатывала план на новый цикл.

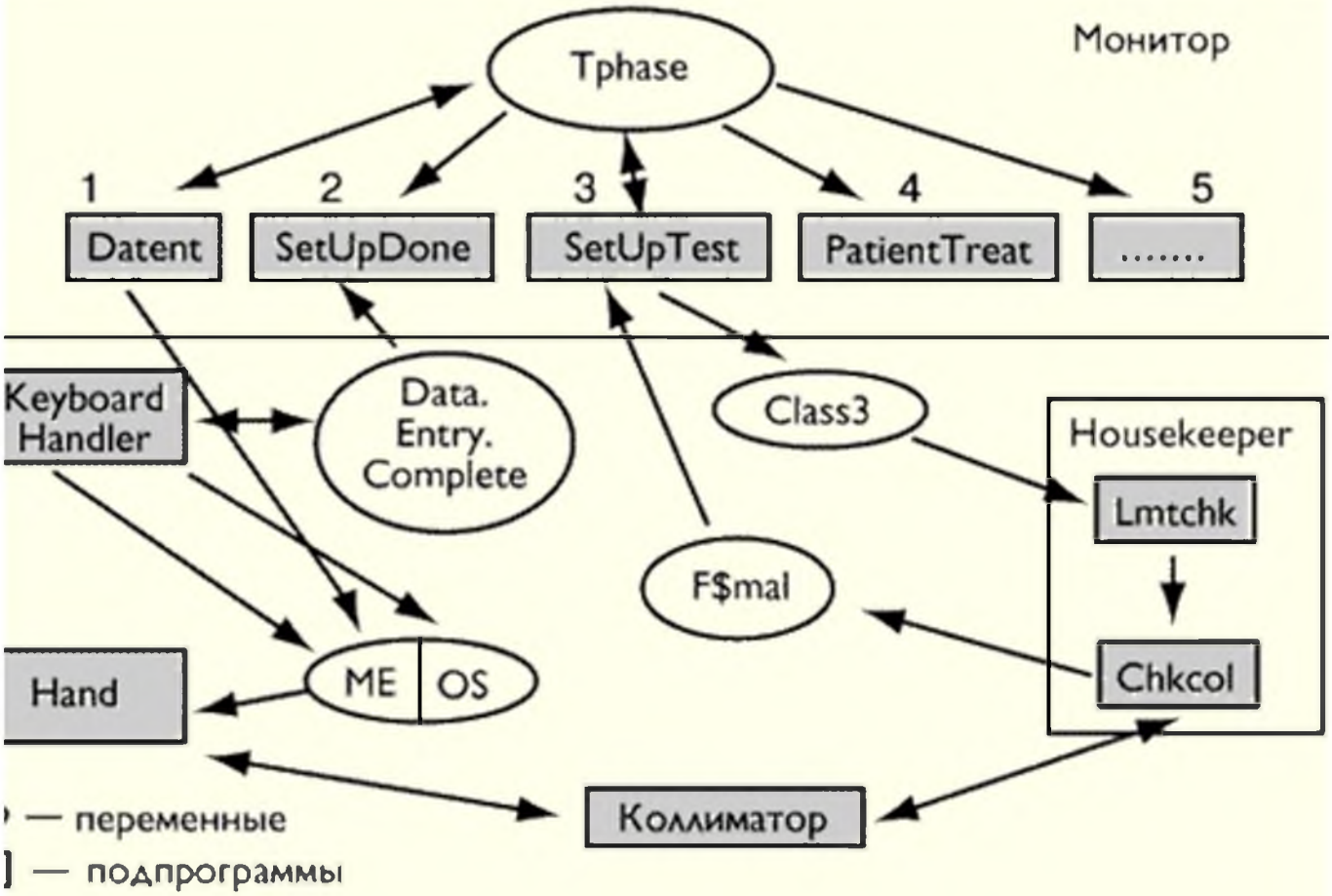

Рис. 1. Взаимодействие задач и подпрограмм в ПО для Therac-25.

Одна из вызываемых Treat подпрограмм Datent (Data entry) через разделяемую «флаговую» переменную Data_entry_complete взаимодействует с «некритической» задачей Keyboard Handler, которая управляет вводом информации с клавиатуры, исполняясь параллельно с Treat. Keyboard Handler распознает момент окончания ввода и сигнализирует об этом, изменяя значение Data_entry_complete. В свою очередь, Datent проверяет значение этой переменной. Если оно не изменилось, то значение Tphase остается равным «1», и на следующем цикле Treat опять запустит Datent; если же значение Data_entry_complete изменилось, то Datent меняет

значение Tphase с «1» на «3»; в результате после окончания работы Datent монитор Treat вызовет подпрограмму Set Up Test, выполняющую проверку считающихся уже установленными параметров.

Необходимо упомянуть еще одну переменную MEOS (Mode/Energy Offset), разделяемую между Datent, Keyboard Handler и еще одной некритической задачей Hand. Старшие байты MEOS используются подпрограммой Datent для установки одного из двух режимов облучения и величины энергии испускаемого потока, в то время как младшие используются параллельно работающей задачей Hand для установки коллиматора в положение, соответствующее выбранному режиму и энергии.

Оператор мог после ввода параметров режима и энергии редактировать эти величины по отдельности. Однако, здесь присутствовал тонкий момент разработчики установили: об окончании процесса ввода (и редактирования!) параметров свидетельствует то, что все параметры заданы и курсор находится в командной строке, на предмет чего каждые 8 сек. (величина выбрана, исходя из некоторых технических соображений, связанных с инерционностью приборов) производится опрос переменной Data_entry_complete. Если в пределах этих 8 сек. курсор покидает командную строку и после быстрого редактирования параметров успевает вернуться на нее, то Keyboard Handler этого события просто не заметит, и соответственно, никак переменную Data_entry_ complete не изменит.

Иными словами, потенциально существует возможность для следующей последовательности действий:

* Keyboard Handler отследил местонахождение курсора на командной строке и установил флаг Data_entry_complete;

* затем оператор изменил данные в MEOS;

* не заметив этого (если к моменту опроса курсор оказался вновь на командной строке), Keyboard Handler не переустанавливает флаг Data_entry_complete;

* тогда Datent уже не способна обнаружить изменение MEOS она свою работу заканчивает, установив Tphase=3 (а не Tphase=1, чтобы отработать еще один цикл и учесть изменения);

* тем временем, параллельно работающая Hand устанавливает коллиматор в положение, соответствующее младшим байтам MEOS (их установила ранее Datent), которые могли находиться в противоречии со старшими байтами этой разделяемой переменной (как раз и подвергшимся редактированию!).

Специальных проверок для обнаружения такой несовместимости предусмотрено не было.

Сноровистая и уже набившая на этой работе руку операторша, в отличие от неторопливых инженеров AECL, скорректировала «режим» и вернула курсор обратно на командную строку очень быстро уложившись в 8 сек. В итоге, проделанное ею изменение режима воспринято не было он остался прежним (рентгеновским), а вот задаваемые параметры (включая находящиеся в младших байтах MEOS, критически

влияющие на величину и направление потоков частиц) соответствовали электронному (фотонному) режиму. Последний штрих в катастрофическую картину внесли показания дозиметра, дававшего показания в «условных единицах» то, что высвеченная «малая» величина дозы относилась к другому режиму и потому не подлежала рациональной оценке, операторше не пришло в голову.

Скорректировать данную ошибку удалось просто введением еще одной разделяемой переменной, которая изменяла значение, как только курсор покидал командную строку. Настоящая беда, однако, заключалась в том, что ошибка такого рода (классическая ошибка, связанная с неправильной синхронизацией одновременно идущих процессов, использующих разделяемые переменные, и приводящая к «race condition») была далеко не единственной.

Программная блокировка и ее последствия

Рассмотрим еще один инцидент с Therac-25, которому суждено было стать последним. Он произошел в Yakima Valley Memorial Hospital (штат Вашингтон) в январе 1987 г. Пациенту было предписано сначало проделать два рентгеновских снимка с дозой в 4 и 3 рад соответственно, а затем произвести в фотонном режиме облучение в 86 рад. Все это и было выполнено, однако, как потом было установлено, пациент получил переоблучение фотонной дозой до 10000 рад.

(Установлено было «потом», а не сразу оператор, сделав снимки, забыл вынуть рентгеновскую пленку из-под пациента, из-за чего у него на консоли горели все

те же 7 рад; однако, и правильная индикация уже выданной дозы была бы здесь как в буквальном смысле слова мертвому припарки).

Что же произошло? Выявленная в итоге расследования проблема выходит далеко за пределы частного случая еще одной программистской ошибки. В данном случае не сработала блокировка, реализованная программно позволившая прибору действовать (испускать поток фотонов) при ошибочной установке параметров.

Ситуация возникла в момент, когда введенные параметры уже верифицированы подпрограммой Datent и монитор Treat в соответствии со значением переменной Tphase = 3 вызвал подпрограмму Set Up Test.

Во время установки и подгонки параметров подпрограмма Set Up Test вызывается несколько сотен раз пока все параметры не будут установлены и верифицированы, о чем эта подпрограмма судит по нулевому значению разделяемой переменной F$mal. Если же значение ненулевое цикл повторяется.

F$mal, в свою очередь, устанавливается подпрограммой Chkcol (Check Collimator) из критической задачи Housekeeper, проверяющей, все ли с коллиматором нормально; а вызывает Chkcol другая подпрограмма задачи Housekeeper под названием Lmtchk (analog-to-digital limit checking), и вызов этот происходит, только если значение разделяемой переменной Class3 ненулевое. А ненулевым его делает как раз сама Set Up Test, которая (пока F$mal=0) каждый раз выполняет над Class3 операцию инкремента.

Эта переменная однобайтовая, следовательно каждый 256-й проход заставляет ее сбрасываться в ноль.

А ведь этот ноль свидетельство, что все параметры, наконец, установлены. Если повезет, что именно в этот момент оператор нажмет клавишу «set» для запуска установки коллиматора в надлежащую позицию (а он это может сделать в любой момент, так как уверен, что система позволит коллиматору начать позиционироваться, только если все параметры заданы и верифицированы), то основываясь на случайно возникшем нулевом значении Class3, подпрограмма Lmtchk уже не станет вызывать Chkcol, а значит установить ненулевое значение F$mal будет некому. Иными словами, в ситуации, когда параметры не установлены должным образом (в данном конкретном случае «челюсти» коллиматора были еще раскрыты слишком широко), программная блокировка не сработала: Set Test Up установила Tphase = 2, что позволило монитору Treat прекратить цикл вызова Set Up Test, а инициализировать подпрограмму Set Up Done, по существу запускающую процесс излучения, который и потек бурным потоком, а не узеньким ручейком, как предполагалось.

Коррекция этой ошибки также выполняется просто вместо выполнения инкремента переменной Class3 следует просто присваивать фиксированное ненулевое значение. Вот от каких, казалось бы, мелких и чисто технических ляпсусов программиста может зависеть жизнь человека!

Некоторые итоги

История с Therac-25 показательна, прежде всего, своей комплексностью: если в случае с Ariane 5 авария случилась один раз и из-за единственной ошибки, то

катастрофические последствия с Therac-25 проявлялись неоднократно в течение длительного времени, и были следствием целого спектра причин, среди которых не только вполне конкретные программистские «баги», но и дефекты в самой постановке выполнявшегося многие годы проекта.

Можно долго перечислять проявившиеся в этом проекте проблемы, например, касающиеся принципов построения человеко-машинного интерфейса (выдаваемые оператору сообщения о критических с точки зрения безопасности ситуациях выглядели как рутинные; при этом не включалась блокировка, препятствующая дальнейшей деятельности оператора и т. д.). Все это является отражением того факта, что как позволяет утверждать ставшая доступной информация о проектных и технологических особенностях разработки квалификация коллектива разработчиков и организация их работы не позволяли реализовать столь сложный и тонкий проект с обеспечением безопасности функционирования, необходимой в данной предметной области.

Что же до системной «глобальной особенности», то к ней можно отнести принципиальную переусложненность построения мультизадачной управляющей системы. С чисто программистской точки зрения можно отметить, что для реализации тонкой синхронизации параллельных процессов был выбран механизм разделения переменных, требующий очень внимательной проработки (это именно та область, где необходимо выполнять формальное доказательство правильности алгоритма, благо соответствующие методы разработаны и могут считаться рутинными для тех, кто ими владеет); получилось же так, что потенциально

опасные в плане возникновения «race conditions» операции типа «set» и «test» не были сделаны «неделимыми» (indivisible), что и привело к наложению друг на друга их «критических секций» и соответственно к печальным последствиям.

Можно выделить и такой фактор, как переоценка уровня безопасности, в принципе гарантируемого программным обеспечением. Это послужило стимулом заменить используемые в предыдущих версиях системы защитные механизмы, которые контролировали радиационные потоки и блокировали их в случае выхода из нормального режима, с «аппаратных» блокираторов (на базе электронно-механических устройств) на чисто программные. Роковую роль сыграло и отсутствие должным образом поставленной системы контроля и исследования природы задокументированных инцидентов, а также некорректные процедуры оценки риска, которые не учитывали специфику ПО. Каждый раз после очередного случая с переоблучением производитель утверждал, что причина выяснена и корректирующие действия предприняты; это не было ложью, но потребовалось два года, чтобы от исправления частностей (которые не делали систему безопаснее) перейти, наконец, к трезвой оценке глобальных особенностей проекта, изменить дисциплину разработки и выполнить корректный анализ рисков.

Мифы о безопасности ПО

Катастрофы с Ariane 5 и Therac-25, сами по себе беспрецедентные, конечно же не являются уникальными. Можно привести длинный список больших и малых инцидентов в системах, относящихся к классу mission-critical, произошедших по причине дефектов в программном обеспечении и проявившихся только в режиме эксплуатации. Конечно, большинство инцидентов так или иначе расследовалось и осмыслялось. К сожалению, специфика «ответственных» систем часто такова, что это осмысление не становилось достоянием всего программистского сообщества поэтому, неудивительно, что в разное время и в разных местах повторялись сходные ошибки. Соответственно, слишком многие приобретают специфические знания и опыт на практике, методом проб и ошибок, которые как лишний раз показывает разобранные инциденты обходятся дорого.

Что же может предложить в этом отношении наука? Только недавно общесистемные и общеинженерные дисциплины «Безопасность Систем» (System Safety) и «Управление Рисками» (Risk Management) начали настраиваться на ту выраженную специфику, которую имеют программно-аппаратные комплексы в контексте их разработки, эксплуатации и сопровождения. Крупнейший специалист в данной области профессор Вашингтонского Университета энси Левесон (Nancy Leveson) ввела даже специальный термин Safeware, который вынесла в название своей книги[8] пока единственной в мировой

[8] N. Leveson «Safeware: System Safety and Computers», Addison-Wesley, 1995

литературе, где систематически рассматриваются вопросы безопасности и рисков в компьютерных системах. В частности, в этой книге разбираются некоторые распространенные мифологические представления о ПО и связанных с ним безопасности и рисках, бытующие на фоне все более широкого использования сложных систем в потенциально опасных приложениях. Остановимся на некоторых из них.

О «дешевом и технологичном» ПО

Бытует мнение, что стоимость программно-аппаратных систем обычно меньше, чем аналоговых или электромеханических, выполняющих ту же задачу. Однако, это миф, если, конечно, не говорить о «голом» hardware и однажды оплаченном ПО, сработанном «на коленке». Стоимость написания и сертификации действительно надежного ПО очень высока; к тому же необходимо принимать во внимание затраты на сопровождение опять же такое, которое не подрывает надежности и безопасности. Показательный пример: только сопровождение относительно простого и не очень большого по объему (около 400 тыс. слов) программного обеспечения для бортового компьютера, установленного на американском космическом корабле типа Shuttle, стоит NASA 100 млн. долл. год.

Следующий миф заключается в том, что ПО при необходимости достаточно просто модифицировать. Однако, и это верно только на поверхностный взгляд.

Изменения в программных модулях легко выполнить технически, однако трудно сделать это без внесения

новых ошибок. Необходимые для гарантий безопасности верификация и сертификация означают новые большие затраты. К тому же, чем длиннее время жизни программы, тем более возрастает опасность вместе с изменениями внести ошибки например, потому, что некоторые разработчики с течением времени перестают быть таковыми, а документация редко является исчерпывающей. Оба примера что с Ariane 5, что с Therac-25 вполне подтверждают эту точку зрения. Между тем, масштабы изменений в ПО могут быть весьма велики. Например, ПО для космических кораблей типа Shuttle[9] за 10 лет сопровождения, начиная с 1980 г., подверглось 14-ти модификациям, приведшим к изменению 152 тысяч слов кода (полный объем ПО 400 тысяч слов).

Необходимость модернизации ПО диктовалась периодическим обновлением аппаратной базы, добавлением функциональности, а также происходило по причине необходимости исправления выявленных дефектов. По оценке независимых экспертов, эти модификации поначалу не сопровождались должными процедурами по поддержке безопасности, однако, случившаяся в 1986 г. авария с кораблем Challenger, которая хотя и произошла по причинам, не связанным с ПО, послужила толчком к пересмотру всей политики NASA в области безопасности, затронув и область ПО.

Наконец, вряд ли справедливо мнение, что все более входящий в практику принцип повторного

---

[9] «An Assessment of Space Shuttle Flight Software Development processes», — Committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Development Processes, National Research Council, 1993

использования ПО дает повышенные гарантии безопасности.

Мысль о том, что использование имеющего длительную историю и уже зарекомендовавшего себя с положительной стороны модуля, равно как и «коробочного» продукта, дает гарантии отсутствия в нем ошибок, весьма естественна с точки зрения «здравого смысла» и способна притупить бдительность. На самом деле повторное использование программных модулей может и понизить безопасность по той простой причине, что данные модули изначально разрабатывались и отлаживались для использования в ином контексте, а спецификация обычно не дает исчерпывающего отчета о всех видах возможного поведения модуля (произошедшая с Ariane 5 авария имеет основной причиной именно повторное использование модуля с некорректной для изменившегося контекста спецификацией).

В случае с Therac-25 большой вклад в произошедшие инциденты внесли модули, изначально разработанные для предыдущей версии системы (Therac-20) во всяком случае, было точно установлено, что именно ошибки в этих повторно-используемых модулях вызвали по крайней мере два смертных случая.

Причем, эти ошибки (как уже было установлено задним числом) проявлялись и при работе Therac-20, но та система была устроена так, что массивного переоблучения не происходило, а потому и процесс коррекции ошибок не запускался.

Можно привести еще несколько любопытных иллюстраций к проблемам, связанным с повторным использованием. Так, попытка внедрить в Англии

программную систему управления воздушным движением, которая до того несколько лет успешно эксплуатировалась в США, оказалась сопряжена с большими трудностями ряд модулей весьма оригинальным образом обращались с информацией о географической долготе: карта Англия уподоблялась листу бумаги, согнутому и сложенному вдоль Гринвичского меридиана, и получалось, что симметрично расположенные относительно этого нулевого меридиана населенные пункты накладывались друг на друга. В Америке, через которую нулевой меридиан не проходит, эти проблемы никак не проявлялись. Аналогично, успешно функционировавшее авиационное ПО, изначально написанное с неявным прицелом на эксплуатацию в северном полушарии, создавало проблемы, когда его стали использовать при полетах по другую сторону экватора. Наконец, ПО, написанное для американских истребителей F-16, явилось причиной нескольких инцидентов, будучи использованным израильской авиацией при полетах над Мертвым морем, которое, как известно, находится ниже уровня моря. Это лишний раз подтверждает мысль, что безопасность ПО нельзя оценивать в отрыве от среды, в контексте которой эксплуатируется вся система.

О «корректном» ПО

Заветная мечта (не столько программистов, сколько потребителей), чтобы в ПО не было ошибок, увы, никак не исполняется. И иллюзий на этот счет уже не осталось. Соответственно, утверждение, что тестирование ПО и/или «доказательство» его корректности позволяют

выявить и исправить все ошибки, можно признать тем мифом, в который мало кто верит.

Причина очевидна. Прежде всего, исчерпывающее тестирование сложных программных систем невозможно в принципе: реально проверить только небольшую часть из всего пространства возможных состояний программы. В результате, тестирование может продемонстрировать наличие ошибок, но не может дать гарантию, что их нет. Как ядовито замечают Жезекель и Мейер,[10] собственно, сам запуск Ariane 5 и явился весьма качественно выполненным тестом; правда, не каждый согласится платить полмиллиарда долларов за обнаружение ошибки переполнения.

Что же касается использования математических методов для верификации ПО в плане его соответствия спецификации, то оно (несмотря на оптимизм, особенно явный в 70-х г.) пока не вошло в практику в сколько-нибудь значительном масштабе, хотя и сейчас некоторые влиятельные специалисты продолжают утверждать, что это непременно случится в будущем. Вопрос, реалистично ли ожидать, что для систем масштаба Ariane 5 возможно выполнить полный цикл доказательства правильности всего ПО, остается открытым. Нет сомнений, однако, что для отдельных подсистем такая задача может и должна ставиться уже приводились аргументы о полезности использования формальных методов при разработке механизмов синхронизации в Therac-25.

Формальные методы разработки это тема специального большого разговора. Здесь же в качестве

---

[10] См. сноску 3

примера формального подхода, имеющего промышленные перспективы, упомянем только «B-Method»,[11] получивший недавно широкое паблисити в связи с созданием ПО для автоматического управления движением на одной из линий парижского метро. Разработчик метода Жан-Раймон Абриал (J.-R. Abrial), до того известный как создатель формального метода Z (вошедшего в учебные программы всех уважающих себя университетов), использовал идеи таких классиков, как Эдсгар Дийкстра (E.W.Dijkstra) и Тони Хоар (C.A.R.Hoare).

Важно, что основанная на формализмах методология поддержана практической инструментальной средой разработки Atelie B (которая, кстати, не единственная).

Эта среда включает в себя инструменты для статической верификации написанных на B-коде компонентов и для автоматического выполнения доказательств, автоматические трансляторы из B-кода в Си и Ада, повторно-используемые библиотеки B-компонентов, средства графического представления проектов и генерации документации, гипертекстовый навигатор и аниматор, позволяющий в интерактивном режиме моделировать исполнение проекта из спецификации, и, наконец, средства по управлению проектом. При разработке ПО для метро, включавшего около 100 тысяч строк B-кода (что эквивалентно 87 тыс. строк на Ада) пришлось доказать около 28 тысяч лемм. Насколько этот подход (и аналогичные ему) будет востребован практикой, покажет будущее.

---

[11] J.-R. Abrial «The B-Book: Assigning Programs to Meanings», //Cambridge University Press, 1996

И все же, такого рода верификация все равно не способна решить все проблемы, в частности, потому, что требуется специфицирование «корректного поведения» программной системы на формальном математическом языке, а это может быть очень непросто. К тому же, источник многих потенциально опасных ошибок может быть не связан непосредственно с вычислительными и алгоритмическими аспектами. Например, в 1992 г. большой резонанс получил произошедший в Англии случай, когда «пошел в разнос» компьютер на станции скорой помощи: причина неожиданно проявившиеся трудности с синхронизацией процессов в условиях большого количества поступивших заявок.

О «надежном» ПО

Теперь о менее очевидном мифе, который звучит так: программно-аппаратные системы обеспечивают заведомо большую надежность по сравнению с теми традиционными (например, электро-механическими) приборами, которые они заменяют.

Понятно, что аппаратные системы способны выдать случайный сбой, могут неправильно реагировать на изменившиеся условия окружающей среды и со временем изнашиваются. К тому же управление ими критически зависит от «человеческого фактора». А вот программное обеспечение ничему этому вроде бы не подвержено, а значит уже поэтому возложение на него функций, до того реализуемых на аппаратном или «операторном» уровне, уменьшает риски и повышает безопасность. И с этим очень хотелось бы согласиться, вот только рассмотренные частные случаи не позволяют вероятность систематических проектных ошибок даже в

программных разработках, выполняемых высококвалифицированными коллективами для требовательных заказчиков, совсем ненулевая.

В конце 80-х гг. такая влиятельная в оборонных кругах организация как British Royal Signals and Radar Establishment сделала попытку оценки распространенности дефектов в ПО, написанном для ряда очень ответственных систем. Оказалось, что «до 10 % программных модулей и отдельных функций не соответствуют спецификациям в одном или нескольких режимах работы».[12]

Такого рода отклонения были обнаружены даже в ПО, прошедшем полный цикл всестороннего тестирования. Хотя большинство обнаруженных ошибок были признаны слишком незначительными, чтобы вызвать сколь-либо серьезные последствия, все же 5 % функций могли оказывать разного рода значимое негативное воздействие на поведение всей системы. Примечательно, что среди прочего авторы исследования особо упомянули выявленную в одном из модулей неназванной системы потенциальную возможность переполнения в целой арифметике, что могло привести к выдаче команды приводу повернуть некую установку не направо (как следовало), а налево. Достаточно предположить, что речь в ПО шла об управлении ориентацией пусковой ракетной установки, чтобы представить возможные последствия.

Коварство программных ошибок и в том, что они могут проявиться далеко не сразу, иногда после сотен тысяч часов нормальной эксплуатации как реакция на

---

[12] См. сноску 8

вдруг возникшую специфическую комбинацию многочисленных факторов. Так, установка Therac-25 вполне корректно работала в течение нескольких лет до первого переоблучения; и последующие зафиксированные инциденты происходили спорадически в течение 2.5 лет на общем «нормальном» фоне. NASA инвестировала огромные средства и ресурсы в верификацию и сопровождение программного обеспечения для космических кораблей Shuttle. есмотря на это, за 10-летие с 1980 г. времени начала использования ПО выявлено 16 ошибок «первой степени серьезности» (способных привести к «потере корабля и/или экипажа»). Восемь из этих ошибок не были обнаружены своевременно и присутствовали в коде во время полетов, хотя, к счастью, без последствий.

Зато во время полетов были задокументированы проблемы, возникшие от проявившихся 12 значимых ошибок, из которых три относились ко «второй степени серьезности» («препятствуют выполнению критически важных задач полета»). А ведь NASA имеет, может быть, самую совершенную и дорогостоящую комплексную систему процессов разработки и верификации ПО.

В то время как надежность аппаратуры может быть увеличена за счет ее дублирования, что резко нивелирует опасности от случайных сбоев, эквивалентного способа защиты от систематических программных ошибок не найдено (даже если некоторые вендоры, с подачи оторванных от практики исследователей, рекламируют методики и инструментарий, позволяющие разрабатывать «zero-defect software»). Впрочем, если бы методы производства идеального ПО существовали, то резонно

предположить, что следование им потребовало бы нереалистично большого количества ресурсов и времени.

Повсеместно, в том числе и при создании ответственных систем, наблюдаемая тенденция свидетельствует о движении в обратном направлении в сторону снижения издержек, и стоимостных, и временных.

Наконец, как ни парадоксально это звучит, даже если бы компьютерные системы действительно были надежнее «традиционных», то это вовсе не обязательно означает, что они обеспечивают большую безопасность. Дело в том, что надежность ПО традиционно определяется степенью его соответствия зафиксированным в спецификациях требованиям; однако, часто бывает так, что ПО делает именно то, что ему и было предписано, и авария Ariane 5 классический тому пример: и злополучное вычисление посторонней для полета величины горизонтального отклонения Инерциальной Платформы, и реакция на него вплоть до выведения из строя всех навигационных систем и бортовых компьютеров все это случилось в полном соответствии с Требованиями, которые были частично унаследованы от Ariane 4 и не отражали новых реальностей.

Более того, по сравнению с ошибками в коде именно спецификационные ошибки обычно ведут к более тяжелым последствиям компетенции разработчиков ПО недостаточно для обнаружения таких ошибок. Программный комплекс сложная система, однако реальный мир, отражаемый в спецификационных требованиях еще более сложен и требует специальных экспертных знаний. Так что надежность ПО и его

безопасность понятия, хотя и перекрывающиеся, но не идентичные.

Фактически любая сложная программная система при определенных обстоятельствах способна вести себя неожиданно для разработчиков и/или пользователей. Вероятность такого поведения, особенно если оно может привести к тяжелым последствиям, следует реалистически оценивать и предусматривать специальные средства защиты, в том числе уже не на уровне самого ПО, а на уровне всей системы. Собственно, авария с Ariane 5 продемонстрировала это в полной мере: реагируй система на выброшенное исключение не столь радикально, аварии бы не произошло ведь сам полет проходил нормально, но этот «глобальный контекст» просто не принимался во внимание!

Аналогично, катастрофические последствия при использовании Therac-25 наступили не столько из-за ошибок, допущенных в ПО, сколько вследствие того, что на аппаратном уровне не было предусмотрено защиты против этих ошибок.

Шлейф программных ошибок тянулся к Therac-25 от ранних версий этого сложного программно-аппаратного комплекса, но в предыдущей модели Therac-20 надлежащие аппаратные защитные механизмы были задействованы от них отказались по соображениям достижения большей производительности. К тому же программных ошибок оказалось много: в каждом конкретном инциденте проявлялась одна из них, ее исправляли затем следующий инцидент (уже со смертельным исходом) показывал, что исправлено не

все. Безопасность это свойство всей системы, а не только ее программного компонента.

Эпитафия

Очевидный урок: приводящие к катастрофическим последствиям дефекты в ПО являются результатом пренебрежения хорошо структурированными и стандартизованными инженерными методами и технологиями, которые, впрочем, должны применяться в контексте контроля всех аспектов разработки и функционирования ответственных систем, включая «человеческий фактор». К сожалению, далеко не всем понятно, что разработка программно-аппаратных систем это именно инженерный процесс, требующий продуманной и поставленной технологии и опирающийся на исполнителей высокой квалификации. Об этом не устают напоминать классики (например, Вирт[13]), но слышат ли их?

На наших глазах повышается сложность программно-аппаратных систем, традиционно не относящихся к разряду mission-critical. е за горами время, когда на массовый потребительский рынок начнут поступать программные комплексы, дефекты в которых могут оказаться крайне неприятными для неготовых к принятию риска «простых» граждан. В конце концов, даже обычный утюг способен вызвать пожар и наверное, какой-нибудь программно-управляемый супер-кухонный-комбайн XXI века может (при ПО надлежащего «качества») повести себя неожиданно (а то и опасно) для домохозяйки.

---

[13] К. Пешио «иклаус Вирт о Культуре Разработки ПО», //Открытые Системы, № 1(27), 1998, сс. 41–44, http://www.osp.ru/os/1998/01/41.htm

Между тем, сегодня на массовом рынке программных продуктов стандарты качества сознательно занижены (о чем мне уже доводилось писать применительно к производственной культуре Microsoft[14]). Торжествует «good enough software», когда критерии качества не имеют столь высокого приоритета, как удобство пользования, простота освоения и дешевизна и все это в сочетании с избыточной функциональностью, пожирающей компьютерные ресурсы, и отсутствием у производителя стимула выбрасывать на рынок действительно отлаженный продукт. Весьма взрывчатая смесь. В обозримом будущем мы можем оказаться свидетелями катастрофических ситуаций, в которые попадут пользователи массовых продуктов, и, похоже, что только это способно сдвинуть ситуацию с ее нынешнего печального положения.

К тому же, функционирующий в соответствии с законами «массовой культуры» рынок обрастает глашатаями того же розлива, обслуживающими тех, кто заказывает музыку. Как и положено в шоу-бизнесе, музыка играет громко и агрессивно. К сожалению, именно в России, по сравнению с западными странами, нарушен баланс между «популярной» компьютерной периодикой и той, что ориентирована на программистов-профессионалов, которым не так просто найти материалы, полезные для совершенствования. В результате размываются критерии, и энтузиасты (которых можно, в зависимости от точки зрения отнести и к «хакерам», и к «чайникам»), освоившие какой-нибудь новомодный инструмент «за 21 день», считают себя

[14] В. Аджиев «MS: Корпоративная Культура Разработки ПО», //Открытые Системы, № 1(27), 1998, с. 45–51, http://www.osp.ru/os/1998/01/45.htm

готовыми к серьезной работе. А если найдутся менеджеры того же пошиба, которые им эту работу действительно доверят?

Небезынтересно в этом контексте упомянуть, что та же Microsoft пытается проникнуть на рынок «ответственных» систем; во всяком случае, она не прочь получить подряд на поставку большой партии Window NT для федеральных организаций, чья работа завязана на «обеспечение национальной безопасности».

А для этого надо сертифицировать эту ОС на соответсвие одному из базовых уровней безопасности компьютерных систем C2 (в соответствии с критериями, определенными в «Оранжевой книге» Агенства ациональной Безопасности США).

Хотелось бы, конечно, надеяться, что осваивая сегмент рынка с принципиально иными требованиями к продуктам, производители массового ПО поднимут планку качества и в своей традиционной вотчине. А ну как окажется наоборот?

И последнее. Все примеры в данной статье относятся к зарубежным системам. В России всегда была очень мощная отрасль mission-critical систем, выглядевшая вполне конкурентоспособно на общем мировом фоне. Отрадно, что и в нынешние турбулентные времена еще сохранились работоспособные коллективы. Однако с авариями самых разных видов и масштабов в России недостатка тоже никогда не было. Например, можно вспомнить о нескольких не столь давних авариях при пусках ракет носителей, внешне очень похожих на катастрофу Ariane 5, да и произошедших примерно в то же время. Что нам известно об их причинах?

Эпилог

Что касается Therac-25, то после выполненной-таки в 1988 году капитальной ревизии комплекса и коррекции ряда проектных и реализационных решений, ни о каких новых инцидентах не сообщалось.

Первый после аварии запуск Ariane 5 несколько раз откладывался и состоялся 30 октября 1997 г. Между тем, летопись катастроф при запусках ракет продолжала пополняться. Так, в августе 1998 г. взорвались при старте ракета Titan-4 (производства Lockheed Martin), выводившая на орбиту американский шпионский спутник (стоимость аварии оценена в 1.2 млрд. долл.), а неделей позже ракета Delta-3 (производитель Boeing), убытки «всего» 225 млн. долл. В начале сентября та же судьба постигла украинскую ракету Зенит, стартовавшую с Байконура. Виновато ли в этих авариях ПО? Может быть, может быть...

Примечания

1

«Ariane 5: Flight 501 Failure», http://www.eIRSn.esa.it/htdocs/tidc/press/Press96/press33.html

2

См. сноску 1

3

J.-M. Jezequel, B. Meyer «Put It in the Contract: The Lessons of Ariane», // Computer, Vol.30, No.2, January 1997, pp.129–130

4

Б. Мейер «Построение надежного объектно-ориентированного ПО. Введение в Контрактное Проектирование», //Открытые Системы, № 6, 1998

5

K. Garlington «Critique of „Put it in the Contract: The Lessons of Ariane"», March 1998 http://www.flash.net/~kennieg/ariane.html

6

B. Nuseibeh «Ariane 5: Who Dunnit?», //IEEE Software, Vol.14, No.3, 1997, pp.15–16

7

N. Leveson, C. Turner «An Investigation of the Therac-25 Accidents», — Computer, Vol.26, N.7, July 1993, p. 18–41

8

N. Leveson «Safeware: System Safety and Computers», Addison-Wesley, 1995

9

«An Assessment of Space Shuttle Flight Software Development processes», — Committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Development Processes, National Research Council, 1993

10

См. сноску 3

11

J.-R. Abrial «The B-Book: Assigning Programs to Meanings», //Cambridge University Press, 1996

12

См. сноску 8

13

К. Пешио «иклаус Вирт о Культуре Разработки ПО», //Открытые Системы, № 1(27), 1998, сс. 41–44, http://www.osp.ru/os/1998/01/41.htm

В. Аджиев «MS: Корпоративная Культура Разработки ПО», //Открытые Системы, № 1(27), 1998, с. 45–51, http://www.osp.ru/os/1998/01/45.htm