Использование формальных методов для обеспечения соблюдения программных стандартов

А. И. Гриневич, В. В. Кулямин, Д. А. Марковцев, А. К. Петренко, В. В. Рубанов, А. В. Хорошилов

Институт системного программирования РАН (ИСП РАН), Б. Коммунистическая, 25, Москва, Россия
E-mail: {tanur,kuliamin,day,petrenko,vrub,khoroshilov}@ispras.ru

Аннотация

В статье описывается подход к построению инфраструктуры использования программных стандартов. Предлагаемый подход основан на формализации стандартов и автоматизации построения тестов для проверки соответствия им из полученных формальных спецификаций. В рамках этого подхода предлагается технологическая поддержка для решения ряда возникающих инженерных и организационных задач, что позволяет использовать его для сложных промышленных стандартов программного обеспечения. Этот тезис иллюстрируется использованием описанного метода для формализации ядра Базового стандарта Linux (Linux Standard Base). Данная работа лежит в рамках подходов к решению задачи по обеспечению развития надежных крупномасштабных программных систем, провозглашенной международным академическим сообществом одним из Больших Вызовов в информатике [1,2].

1. Введение

Экономическое и социальное развитие общества требует для своей поддержки все более сложное программное обеспечение (ПО). Масштабность современных программных систем приводит к тому, что такие системы строятся из многих компонентов, разрабатываемых в разных организациях огромным количеством людей. Для построения работоспособных систем на основе такого подхода необходимо уметь решать задачи обеспечения совместимости между различными их компонентами и их надежности в целом. Одно из наиболее эффективно работающих на практике решений этой задачи — выработка и соблюдение *стандартов на программные интерфейсы* между отдельными компонентами.

Идея, лежащая в основе использования интерфейсных стандартов, проста — настаивая на их соблюдении, мы обеспечиваем компонентам, созданным разными разработчиками, возможность взаимодействовать через стандартизованные интерфейсы. Таким образом, их совместимость обеспечивается без введения чересчур жестких ограничений на возможные реализации, что ограничило бы как творчество отдельных разработчиков, так и инновационный потенциал компаний- поставщиков ПО. Этот подход хорошо работает,

если стандарт определяет функциональность, реализуемую фиксируемым им интерфейсом, достаточно точно и недвусмысленно.

Однако, тексты многих современных стандартов, описывающих требования к программным интерфейсам, далеко не так точны. Это объясняется историей их появления. Обычно такие стандарты разрабатываются под давлением требований рынка и должны принимать во внимание противоречивые интересы многих поставщиков программного обеспечения, интерфейс которого предполагается стандартизовать. В таких условиях только базовая функциональность может быть определена непротиворечивым образом, а для сложных и специфических функций каждая группа разработчиков предлагает свое собственное решение. Поскольку многие поставщики ПО уже вложили деньги в имеющиеся у них решения, им очень тяжело отказаться от этих инвестиций в пользу варианта одного из конкурентов, который при этом остается в выигрыше.

Чаще всего группа по выработке стандарта приходит к компромиссу, при котором основные поставщики должны внести примерно одинаковые по объему изменения в ПО каждого из них, а в тех случаях, где необходима серьезная переработка, стандарт осознанно делается двусмысленным, чтобы любое из уже имеющихся решений могло декларировать соответствие ему.

Это позволяет поставщикам ПО потратить приемлемые для них деньги на обеспечение соответствия своих систем стандарту, но в то же время подрывает столь желаемую совместимость различных его реализаций. Некоторые из действующих стандартов на интерфейсы ПО или на телекоммуникационные протоколы появились еще 20-25 лет назад и неоднократно пересматривались. Конечно же, каждая новая их версия становилась более строгой и непротиворечивой, и большинство проблем первых выпусков уже устранено. Однако неясности и противоречия постоянно вносятся в новых добавлениях, которые так и будут появляться в связи с постоянным развитием технологий.

Выход из этого замкнутого круга многие специалисты по программной инженерии видят в *формализации требований стандартов.* Сама попытка переформулировать их в строгом виде вскрывает противоречия и двусмысленности. Формализация большого текста, конечно, требует приложения колоссальных усилий, но она может быть проведена не в один прием, а постепенно, инкрементально. Никто также не ожидает, что все неточности удастся удалить из стандарта одним махом. Начав процесс формализации, мы, по крайней мере, получаем информацию об имеющихся реальных проблемах стандарта и можем предлагать и анализировать различные варианты их решения.

Формализация делает стандарт более точным, а значит, и более полезным, но она одна все же недостаточна для адекватного обеспечения соблюдения стандарта на практике. Разработчики ПО нуждаются во вспомогательных инструментах, которые позволят им убедиться в соответствии или несоответствии той или иной системы стандарту. Поэтому, практически полезное формальное

описание требований стандарта должно дополняться набором тестов, которые проверяют соответствие этим требованиям.

Эта статья представляет подход к формализации стандартов и разработке тестовых наборов, основанный на одной из наиболее детально проработанных методик тестирования программного обеспечения на соответствие стандартам, созданной для использования в разработке телекоммуникационного ПО и зафиксированной в стандартах ISO 9646 [3] и ITU-T Z.500 [4].

Продуманная методика делает тестирование на соответствие стандарту более строгим и тем самым обеспечивает более высокое качество ПО, реализующего его. Однако практическое использование наборов тестов на соответствие порождает ряд инженерных и организационных вопросов, без решения которых невозможно перенести применение этой методики для небольших примеров на современные стандарты, описывающие очень сложные системы. Можно отметить следующие аспекты.

- *Прослеживаемость требований.* Для разработчиков ПО и их руководителей подлинным источником требований является сам текст стандарта. Формальные спецификации же — это какой-то дополнительный документ, который должен явно продемонстрировать свою связь со стандартом. То же самое относится и к тестам, полученным из формальных спецификаций — они должны соотноситься с требованиями, описанными в каких-то разделах стандарта. Прослеживаемость требований стандарта в тестах является практической потребностью всякой осмысленной деятельности по их реализации.
- *Покомпонентное рассмотрение стандарта.* Промышленные стандарты иногда очень сложны и часто имеют большой объем, так же как и промышленные программные системы. Для того чтобы адекватно анализировать их, необходимы какие-то техники декомпозиции. Формализм, используемый для спецификаций, должен позволять выделять в стандарте компоненты разных уровней, и анализировать их по отдельности. Должна поддерживаться также и последующая интеграция таких компонентов в единое целое.
- *Управление изменениями.* Стандарты, как и их реализации, не пребывают в неизменном виде — они постоянно изменяются под давлением общего технологического прогресса. Необходима адекватная поддержка изменений в используемом формализме, методах построения тестов и инструментах. Ее отсутствие быстро сделает полученные спецификации и тесты практически бесполезными.
- *Управление конфигурациями.* Очень часто стандарты, используемые на практике, описывают системы с большим числом параметров–конфигурационных опций, значительно влияющих на поддерживаемую функциональность. Возможность выбора конфигурации и проверки только относящихся к ней требований также должна быть заложена в спецификации, тесты и методику их разработки.

Все перечисленные проблемы должны иметь удобные решения в рамках технологии построения и эксплуатации инфраструктуры для обеспечения соблюдения стандартов. В данной статье рассказывается о подходе, претендующем на роль такой технологии и основанном на методе автоматизированной разработки тестов UniTesK, созданном в ИСП РАН. Основные идеи и техники этого подхода рассматриваются в следующих двух разделах статьи. Четвертый раздел представляет предварительные результаты его применения к Базовому стандарту Linux (Linux Standard Base) [5], промышленному стандарту на интерфейс базовых библиотек операционной системы Linux. В последних двух разделах статьи содержится краткий обзор других подходов к построению тестов на соответствие стандартам и заключение, формулирующее основные результаты и перечисляющее направления возможного развития предложенного подхода.

2. Формализация стандартов

Основные трудности, возникающие при формализации индустриальных стандартов, связаны с их неформальной природой. Главные цели такого стандарта — зафиксировать консенсус ведущих производителей относительно функциональности рассматриваемых систем и обеспечить разработчиков реализаций стандарта и приложений, работающих на основе этих реализаций, справочной информацией и руководством по использованию описанных функций. Форма и язык стандарта выбираются так, чтобы достигать этих целей.

Стандарты на программные интерфейсы чаще всего состоят из двух частей: обоснования (rationale), представляющего целостную картину основных концепций и функций в рамках данного стандарта, и справочника (reference), описывающего каждый элемент интерфейса отдельно. В дополнение к элементам интерфейса (типам данных, функциям, константам и глобальным переменным) справочник может описывать и более крупные компоненты: подсистемы, заголовочные файлы и пр. Отдельные разделы справочника могут ссылаться друг на друга и содержать части друг друга.

Формализация стандарта включает несколько видов деятельности.

1. ***Декомпозиция стандарта.*** Так как количество интерфейсных элементов, описанных в стандарте, может быть очень велико, то на первом шаге они разбиваются на логически связанные группы. Такая группа обычно состоит из операций и типов данных, связанных с определенным набором близких функций, и замкнута относительно обращения операций. Например, операции открытия и закрытия файлов, создания и удаления объектов нужно помещать в одну группу. Вся дальнейшая работа производится в рамках таких групп.
2. ***Выделение требований.*** Следующая задача заключается в выделении всех требований стандарта к описываемым им элементам интерфейса, которые могут быть проверены. Так как одна и та же вещь может быть описана в нескольких местах, то все соответствующие разделы стандарта необходимо внимательно, фраза за фразой, прочитать и пометить все найденные

требования и ограничения. Затем выделенные требования объединяются в непротиворечивый набор. Эта работа достаточно утомительная и тяжелая, но ни в коем случае не механическая. Ее сложность связана с преобразованием неформальных требований и ограничений в формальные, а, по словам М. Р. Шуры-Буры, переход от неформального к формальному существенно неформален.

Утверждения и ограничения из разных частей стандарта могут быть несовместимыми, двусмысленными, представлять похожие, но не одинаковые идеи. Только текста стандарта часто не достаточно, чтобы прийти к непротиворечивой полной картине. Поэтому при выделении требований необходимо пользоваться общими знаниями о предметной области, книгами и статьями по тематике, знанием о используемых в существующих системах решениях, а также общаться с авторами стандарта, экспертами в области и опытными разработчиками.

Некоторые аспекты специально не определяются стандартом точно, для того чтобы реализации различных производителей соответствовали ему. Занятный пример можно найти в текущей версии стандарта POSIX [6]. Описание функций `fstatvfs()` и `statvfs()` из `sys/statvfs.h` говорит: «Не специфицировано, имеют ли все члены структуры `statvfs` смысл для всех файловых систем» ("It is unspecified whether all members of the `statvfs` structure have meaningful values on all file systems").

Существует два возможных пути разрешения подобных ситуаций.

- Можно ничего не проверять. В этом случае никаких требований из соответствующей части стандарта не извлекается.
- Если существует небольшое количество возможных реализаций, то требования к ним могут быть представлены в виде параметризированных ограничений. При этом вводится конфигурационный параметр, разные значения которого соответствуют различным возможным реализациям и определяют конечный вид проверяемого требования.

Пример из описания функции `basename()` в стандарте POSIX: «Если строка, на которую указывает параметр `path`, равна “//”, то реализация может возвращать как “/”, так и “//”» ("If the string pointed to by `path` is exactly “//”, it is implementation-defined whether “/” or “//” is returned").

Выделение требований проводится до тех пор, пока весь текст стандарта, касающийся выбранной группы элементов интерфейса, не будет разбит на требования, которые можно проверить, и остальные фразы, не содержащие проверяемых утверждений. Основные результаты этой работы следующие.

- *Каталог требований.* Он состоит из списка проверяемых требований, налагаемых стандартом, и привязывает каждое требование к

соответствующему месту в тексте стандарта. Одно требование обычно соответствует логически замкнутому фрагменту текста, выражающему одно ограничение на элемент интерфейса или элемент данных. Каждое требование имеет уникальный идентификатор. Каталог требований позволяет в дальнейшем оценивать адекватность тестирования в терминах исходного текста стандарта.

- *Размеченный текст стандарта.* Это исходный текст стандарта, в котором наглядным образом, например цветом, выделены места, соответствующие требованиям из каталога. Размеченный текст позволяет проверить полноту выполненного анализа текста стандарта и убедиться в том, что ни одно требование не пропущено.
- *Дефекты стандарта и замечания к его тексту.* Это список обнаруженных двусмысленностей, несоответствий между утверждениями в различных частях стандарта, ненамеренно неясных и неточных ограничений, неполных описаний функциональности и т.д. Этот список передается группе разработчиков стандарта, и позволяет сделать следующие его версии более точными и непротиворечивыми.

3. ***Разработка спецификаций.*** Эта работа обычно выполняется параллельно и вперемешку с предыдущей. Они разделены только в целях ясности изложения.

   Большинство найденных требований записываются в форме *контрактных спецификаций* операций, типов данных и элементов данных. Каждая операция описывается с помощью предусловия и постусловия. *Предусловие* операции определяет область ее определения. *Постусловие* определяет ограничения на результаты работы операции и итоговое состояние системы в зависимости от значений ее входных параметров и состояния системы до ее вызова. Для типов данных и элементов данных определяются ограничения целостности, называемые *инвариантами*.

   Те требования, которые не оформляются в виде контрактных спецификаций, делятся на следующие группы.

   - *Непроверяемые требования*. Часть требований не проверяется, потому что их вообще нельзя проверить с помощью конечной процедуры тестирования, или потому что не существует методов надежной проверки таких требований (например, «функция должна выдавать результат за время, меньшее, чем любой многочлен второй степени от значения ее параметра»), а также из-за крайней неэффективности их проверки («возвращаемый функцией указатель должен отличаться от указателя на любой объект, присутствовавший в системе до ее вызова»).
   - *Проверяемые требования.* Другая часть требований проверяется не в спецификациях по соображениям эффективности тестов и удобства их модификации. Ряд требований, касающихся целостности данных

некоторых типов, может быть отражен в самом описании их структуры в спецификациях. При этом проверка соответствующих ограничений происходит при преобразовании из данных реализации в модельные данные. Если эти ограничения нарушаются, такое преобразование становится невозможным и спецификация не может с ними работать, но сообщение об их проверке и обнаруженном несоответствии появится в трассе теста.

Некоторые требования можно проконтролировать только в результате многократных обращений к тестируемой функции и проверки каких-то свойств всей совокупности полученных при этом результатов. Эти требования часто неудобно оформлять в виде спецификаций — для них создается отдельный тестовый сценарий (см. ниже), который производит все нужные обращения и проверяет необходимые свойства полученных результатов. Примером такой ситуации может служить требование к функции `rand()` генерировать при многократных обращениях последовательность случайных чисел, равномерно распределенных на некотором интервале. Для проверки этого свойства можно накопить данные о результатах ее вызовов и применить к их совокупности, например, критерий Колмогорова.

Разработка спецификаций имеет два результата.

- *Формальные спецификации* всех элементов интерфейса. Код спецификаций размечается, чтобы указать связь между описанными формальными ограничениями и соответствующими им требованиями из каталога требований.
- *Конфигурационная система стандарта.* Эта система состоит из набора конфигурационных параметров, как объявленных в стандарте, так и дополнительно введенных разработчиками спецификаций, зависимостей между ними, а также связей между ними, элементами интерфейса и проверяемыми ограничениями. Некоторые конфигурационные опции, представленные как значения параметров, управляют набором проверяемых требований. Другие могут говорить о том, что определенная функциональность вообще отсутствует в системе, и соответствующие операции не должны вызываться. Третьи могут влиять на возможные коды ошибок, возвращаемые операциями.

  Пример первого случая можно увидеть в описании функции `pthread_create()` из стандарта POSIX: «Если макрос `_POSIX_THREAD_CPUTIME` определен, то новый поток должен иметь доступ к часам процессорного времени, и начальное значение этих часов должно быть выставлено в ноль» ("If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero").

4. ***Определение критериев покрытия.*** Последний вид деятельности в рамках формализации стандарта заключается в определении критериев покрытия, которые могут быть использованы для измерения адекватности тестирования реализации на соответствие стандарту. Базовый критерий — это необходимость покрытия всех требований стандарта, применимых к текущей конфигурации тестируемой системы. Более строгие критерии могут определять дополнительные ситуации для тестирования. Все эти критерии базируются на спецификациях, представляющих требования стандарта.

   Критерии покрытия сложным образом связаны с конфигурационными параметрами. Возможность покрытия определенной ситуации может зависеть от текущих значений конфигурационных параметров, и эта зависимость должна быть зафиксирована, чтобы избежать многочисленных трудностей при анализе результатов тестирования.

   Результатом этой работы является *набор критериев покрытия*, тесно связанный с конфигурационной системой.

Процесс формализации стандарта и последующей разработки тестов на основе ее результатов проиллюстрирован на Рис. 1.

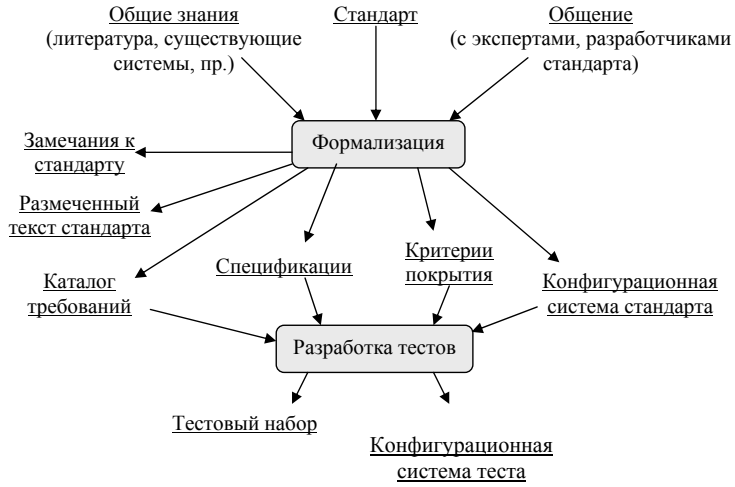

Рисунок 1. Входные и выходные данные формализации стандарта и разработки тестов.

3. Разработка тестов на соответствие стандарту

В основе процесса разработки тестов на соответствие стандарту лежит технология UniTesK, разработанная в ИСП РАН на базе многолетнего опыта проектов по тестированию сложного промышленного ПО. Эта технология использует подход к формальному тестированию, разработанный в работах Берно (Bernot) [7], Бринксмы (Brinksma) и Тритманса (Tretmans) [8,9] и описанный в телекоммуникационных стандартах ISO 9646 [3] и ITU-T Z.500 [4]. Основные положения этого подхода можно сформулировать следующим образом (Рис. 2 иллюстрирует связи между упоминаемыми ниже понятиями).

- Требования стандарта представлены в виде модели, описанной с помощью некоторого формализма. Эта модель называется *спецификацией*.
- Предполагается, что программная система, чьё соответствие стандарту мы пытаемся установить — *тестируемая система,* — может быть адекватно смоделирована с помощью этого же формализма. Соответствующую нашей системе модель мы называем *реализацией*. Мы не знаем её деталей, но предполагаем, что она существует. Адекватность моделирования в данном случае означает, что мы не можем наблюдать никаких различий между реальным поведением тестируемой системы и модельным поведением реализации.
- Тот факт, что тестовая система соответствует стандарту, моделируется посредством *отношения соответствия* (conformance relation, implementation relation) между реализацией и спецификацией.
- Модельные тесты строятся на основе спецификации или извлекаются из нее. *Модельный тест* — это модель в том же самом формализме, способная взаимодействовать с другими моделями и выдающая булевский вердикт как результат этого взаимодействия. Реализация *проходит* модельный тест, если он выдаёт вердикт «истина» в результате взаимодействия с ней. Имеет смысл строить только *значимые тесты*, т.е. те, что проходятся любой реализацией, соответствующей спецификации. Тест, не являющийся значимым, может отвергнуть совершенно корректную реализацию. Результатом извлечения тестов является набор модельных тестов или модельный тестовый набор. Желательно, чтобы он был *полным тестовым набором* — таким, что реализация проходит все тесты из него тогда и только тогда, когда она соответствует спецификации.
- Модельные тесты транслируются в тестовые программы, взаимодействующие с тестируемой системой. Поскольку мы считаем, что это взаимодействие моделируется взаимодействием между модельным тестом и реализацией, можно сделать вывод, что тестируемая система, проходящая полный тестовый набор, соответствует стандарту.

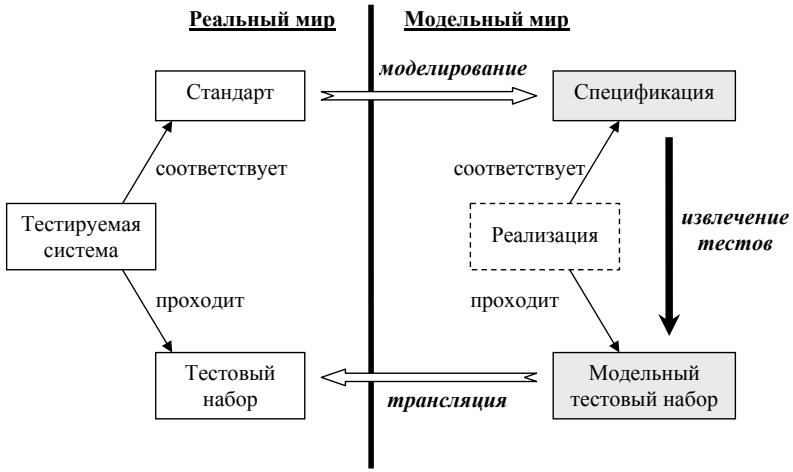

Рисунок 2. Отношения между реальными сущностями и их моделями.

Используемые на практике стандарты обычно описывают довольно сложные и большие системы. Спецификации таких стандартов также сложны и объёмны, так что любой полный тестовый набор для такой спецификации содержит бесконечное множество тестов. Для того чтобы сделать формальное тестирование осуществимым на практике, мы добавили в описанный базовый подход понятие критерия тестового покрытия и нацеленность набора тестов на достижение критерия.

- *Критерий тестового покрытия* — это некоторое отношение эквивалентности на модельных тестах. Фактически, выбор такого критерия означает, что все равно, какой из эквивалентных тестов выполнять — все они должны давать одинаковые результаты. Поэтому, используя критерий тестового покрытия, мы выдвигаем гипотезу, что реализация обязана либо проходить оба эквивалентных теста, либо не проходить ни один из них. Обычно критерий стараются выбрать так, чтобы большинство разумных реализаций обладало этим свойством.

  Тестовый набор называется *полным относительно критерия покрытия,* если объединение классов эквивалентности по этому критерию тестов из этого набора есть полный тестовый набор в определенном выше смысле.

Следует обратить внимание, что критерий покрытия может быть выбран исходя из различных соображений. Единственное обязательное свойство —

наличие конечного тестового набора, полного относительно избранного критерия. При построении тестового набора для проверки на соответствие стандарту разумно использовать критерии, построенные на основе текста стандарта и структуры спецификаций, являющихся его формальным представлением.

Технология UniTesK детально рассматривалась в работах [10-13]. Здесь мы приводим только краткое описание ее основных идей.

- Функциональные требования к поведению тестируемой системы представляются в виде *контрактных спецификаций*. Контракт для отдельного компонента состоит из пред- и постусловий для всех его операций и асинхронных событий, которые он может создавать, и инвариантов, определяющих ограничения целостности его внутренних данных.
- *Критерии тестового покрытия* определяются на основе структуры постусловий операций отдельного компонента или нескольких компонентов. Примерами таких критериев являются покрытие ветвей в коде постусловия и покрытие дизъюнктов из дизъюнктивной нормальной формы условий этих же ветвлений. Есть возможность вводить произвольные критерии, описывая дополнительные ситуации, в которых работу системы необходимо проверить.
- *Тестовый сценарий* создается для достижения определённого покрытия заданной группы операций. Такой сценарий определяет конечный автомат, моделирующий поведение тестируемого компонента таким образом, что выполнение всех его переходов гарантирует покрытие 100% ситуаций в соответствии с выбранным критерием. Автомат задается при помощи функции вычисления состояния и набора допустимых действий в произвольном состоянии. Этот набор действий зависит от данных состояния. Каждое действие соответствует вызову одной из операций с некоторыми аргументами.
- Сами тесты или последовательности тестовых воздействий генерируются при выполнении тестового сценария, во время которого автоматически строится некоторый путь, покрывающий все переходы описываемого им автомата.

Построение тестового набора для проверки соответствия практически используемым стандартам ПО приводит к необходимости разработки *конфигурационной системы тестов*. Эта система включает конфигурационную систему стандарта (см. выше) и дополнительные параметры, управляющие работой тестов. Разные значения этих параметров соответствуют различным критериям покрытия, разным наборам тестовых сценариев и т.п. Хорошая конфигурационная система делает тестовый набор применимым в любых условиях, где может функционировать реализация рассматриваемого стандарта.

4. Применения описанного подхода

Представленный выше подход был успешно использован для формализации и создания наборов тестов для частей стандартов на протоколы IPv6 и IPMP2 [11,14].

Более масштабным применением этого подхода стал проект Центра верификации ОС Linux [15] по формализации Базового стандарта Linux (Linux Standard Base, LSB) [5] и разработке тестового набора для проверки соответствия ему. Проект был назван OLVER — Open Linux Verification. Цель первой фазы проекта — создать формальные спецификации и соответствующий тестовый набор для 1532 функций основных библиотек Linux, перечисленных в разделах III и IV стандарта LSB 3.1 (они описывают базовые библиотеки и библиотеки утилит).

Стандарт LSB задает требования ко многим из этих функций, ссылаясь на уже существующие стандарты. Непосредственно в тексте LSB описывается лишь 15% интерфейсов, большинство — 60% функций — определяются в стандарте Single UNIX Specification [6], который включает текущую версию POSIX, а остальные — в таких спецификациях как X/Open Curses [16], System V Interface Definition [17] и ISO/IEC 9899 (стандарт языка C) [18]. LSB не включает в себя все эти стандарты, а ссылается лишь на их части, касающиеся описания требований для отдельных функций. Всего, вместе со стандартами, на которые он ссылается, LSB состоит из более чем 6000 страниц текста.

Первая фаза проекта завершится в конце 2006 года. Все ее результаты будут доступны в виде открытого кода на сайте проекта [15]. В составе этих результатов будет следующее.

- Дополнения к стандарту LSB Core 3.1 в виде формальных спецификаций его требований на расширении языка C (SeC — Specification Extension of C). Такие спецификации выражают требования стандарта в формальном виде, позволяя прояснить нечеткие места в его тексте и сделать явными все подразумеваемые в нем ограничения. Они послужат основой для разработки тестового набора для проверки соответствия стандарту LSB.
- Список замечаний к текстам LSB и связанных стандартов, наработанных в ходе формализации. Это указания на нечеткие, двусмысленные, противоречивые или ошибочные места в стандартах. Они будут направляться разработчикам стандартов для внесения изменений в их будущие версии.
- Тестовый набор для проверки соответствия поведения системных интерфейсов конкретной реализации Linux требованиям стандарта LSB Core 3.1. Он будет представлять собой набор программ, которые в результате своего выполнения строят HTML-отчеты о проверенных требованиях и найденных несоответствиях. Тестовый набор будет иметь систему конфигурации, которая позволит настраивать тесты на особенности конкретной тестируемой системы, допускаемые стандартом, а также установить параметры, определяющие состав тестируемых модулей и глубину тестирования.

В начале проекта 1532 функции LSB были разбиты на 170 групп функционально связанных операций, которые в свою очередь группируются в большие подсистемы в соответствии с общей функциональностью — потоки, межпроцессное взаимодействие, файловая система, управление динамической памятью, математические функции, и т.д.

За первые 3 месяца работы были проспецифицированы и снабжены базовыми тестовыми сценариями 320 функций из 41 группы. В процессе формализации было выделено около 2150 отдельных требований стандарта к этим функций. При этом к некоторым функциям предъявляется всего лишь несколько требований, в то время как к другим — несколько десятков.

На 1 июня 2006 года проанализирован текст стандарта, касающийся 930 функций. Выделено 10500 элементарных требований. Из этих функций для 740 разработаны формальные спецификации и тесты. Полученный на это момент набор спецификаций и тестов содержит около 400 тысяч строк кода. Текущее состояние проекта отражается на сайте [15].

Качество разрабатываемых тестов определяется исходя из покрытия требований, но сравнить по этой характеристики несколько различных тестовых наборов для тестирования функциональности достаточно тяжело, в них почти всегда элементарные требований выделяются по-разному (см. далее). Если же сравнивать покрытие кода для некоторых групп функций, можно отметить, что полученный в этом проекте тестовый набор дает большее покрытие, чем другие известные наборы для тестирования функциональности. Получаемое покрытие кода приближается к результатам отладочного тестового набора для библиотеки glibc [19], созданного ее разработчиками, обладающими детальными знаниями об особенностях реализации различных ее функций, и не нацеленного на проверку соответствия каким-либо стандартам. Результаты сравнения покрытия кода для некоторых групп функций для тестовых наборов LTP [20], официального тестового набора LSB [21], отладочного тестового набора glibc [19] и набора, полученного в рассматриваемом проекте, приведены в Таблице 1.

Группа функций	LTP [20]	LSB TS [19]	glibc TS [21]	OLVER [15]
Управление потоками	48%	71%	78%	72%
Преобразования строк	53%	67%	84%	91%
Функции поиска данных	26%	33%	70%	65%

Таблица 1. Сравнение получаемого покрытия кода для некоторых групп функций.

На основе достигнутой производительности можно утверждать, что первая фаза проекта потребует около 15 человеко-лет. Чтобы начать работать в этом проекте, опытному разработчику программного обеспечения (не обычному тестировщику!) требуется около месяца для обучения особенностям технологии и изучения

сопутствующих процессов. Полученные предварительные результаты позволяют надеяться, что рассмотренный подход может успешно применяться для проектов такого масштаба.

5. Другие подходы к построению тестов на соответствие стандартам

Наиболее глубокие результаты в области формализации стандартов и методик разработки тестов на основе формальных спецификаций получены применительно к телекоммуникационным протоколам. Общий подход к формальному тестированию, представленный в работах [4,7,9] (см. также выше), тоже был разработан в этой области.

Этот подход имеет много общего с представленным в данной статье. Различия связаны, в основном, с необходимостью иметь дело со стандартами большего объема, такими как стандарты POSIX или LSB на интерфейсы библиотек операционных систем. Большой размер стандартов и соответствующих спецификаций приводит к практической невозможности использовать как полные тестовые наборы в терминах работы [9], поскольку они бесконечны, так и выбор набора целей тестирования «на глаз», который является одним из традиционных шагов разработки тестов для телекоммуникационных протоколов. Вместо этого используется более систематическое выделение отдельных требований, понятие критерия тестового покрытия, выбор критерия, ориентированного на учет всех выявленных требований, и генерация тестов, нацеленная на достижение высоких показателей покрытия по выбранному критерию.

Использование контрактных спецификаций также способствует большей практичности и масштабируемости нашего подхода. Программные контракты, с одной стороны, позволяют провести декомпозицию большой системы на более обозримые компоненты, что труднее сделать, используя автоматы или системы переходов, лежащие в основе традиционного формального тестирования. С другой стороны, пред- и постусловия лучше подходят для описания недетерминированного поведения, которое довольно часто вынужден фиксировать стандарт при наличии нескольких возможностей реализовать одну и ту же абстрактную функциональность.

Статья [22] представляет другую попытку формализации стандарта на примере IEEE 1003.5 — POSIX Ada Language Interfaces (интерфейсы POSIX для языка Ada). В рамках описываемого в ней метода на базе требований стандарта сразу строились формальные описания проверяющих их тестов, без промежуточных спецификаций самих требований. Такой метод кажется нам принципиально мало отличающимся от традиционной ручной разработки тестов, при которой разработчик теста читает стандарт, придумывает на основе прочитанного тесты и записывает их в виде некоторых программ.

Более близкий к рассматриваемому в данной статье подход используется в работе [23], где представлены результаты практического использования инструмента

автоматизированного создания тестов GOTCHA-TCBeans. Этот инструмент использовался, в частности, для построения тестов для функции `fcntl()` стандарта POSIX. По ряду идей, касающихся использования формальных техник для тестирования, представленный в [23] подход очень похож на используемый нами, хотя он более сфокусирован на технике создания формальных моделей, чем на обеспечении их адекватности и поддержке прослеживаемости связей между формальными спецификациями и исходным текстом стандарта. Кроме того, используемые в [23] формальные спецификация являются описаниями конечных автоматов на специализированном языке Murφ, а не программными контрактами, как в нашем подходе.

Существует ряд аналогичных работ по разработке тестовых наборов для проверки соответствия стандартам интерфейсов операционных систем. Наиболее известные стандарты в этой области это IEEE Std 1003.1, или POSIX [6], и Базовый стандарт Linux, или LSB [5]. Имеются и наборы тестов на соответствие этим стандартам — это сертификационные тесты POSIX от Open Group [24], открытый проект Open POSIX Test Suite [25], и официальные сертификационные тесты на соответствие LSB [21] от Free Standards Group.

Все эти проекты используют схожие технологии выделения требований из текста стандарта и создания соответствующего каталога требований. После этого тесты разрабатываются вручную на языке C с применением подхода «один тест на одно требование». Они не используют формализацию требований и автоматическую генерацию тестов.

Стоит отметить, что использование подхода «один тест на одно требование» создает предпосылки для укрупнения требований при их выделении, так как велик соблазн проверить как можно больше в одном тесте. К примеру, в тестах Open POSIX мы обнаружили требования, которые включают в себя десяток утверждений, которые в проекте Центра верификации ОС Linux выделяются в отдельные требования. Такое укрупнение требований может приводить к тому, что некоторые утверждения стандарта упускаются из виду и не проверяются, в то время как крупное интегральное требование, в которое они входят, считается протестированным. В результате построенный на основе таких требований отчет об их покрытии может искажать реальное положение дел, утверждая, что все требования стандарта проверены.

Кроме того, автоматическая генерация тестов из спецификаций, используемая в нашем подходе, делает тестовый набор более управляемым, позволяя описывать сами требования стандарта в одном месте, а технику, используемую для их проверки, и тестовые данные — в другом. Это значительно облегчает внесение изменений в тесты при развитии стандарта или их адаптации под требования специфической предметной области.

6. Заключение

Внедрение и обеспечение соблюдения стандартов на программные интерфейсы связаны с большим количеством сложных проблем, как технического

характера, так и экономических и социальных. Тем не менее, все эксперты сходятся на том, что эта деятельность необходима для стабильного развития производства программного обеспечения. Формализация стандартов уже неоднократно предлагалась в качестве возможного решения множества технических задач, связанных с ней.

Однако формализация требует огромных усилий, что позволяет усомниться в применимости подходов на ее основе к реально используемым программным стандартам, достаточно объемным и сложным. Устранить эти сомнения помогут только практические примеры применения подобных методов к таким стандартам. Упомянутый выше проект [15] по формализации LSB похоже, является одной из первых попыток формализации значительной части используемого на практике стандарта высокой сложности и позволяет почувствовать все ее выгоды и недостатки.

Мы считаем, что представленный в этой статье подход позволит успешно справляться с подобными задачами. Аргументами в пользу этого мнения являются стабильное продвижение описанного проекта, история применений технологических составляющих данного подхода на практике ([12,14]) и лежащие в его основе базовые принципы хорошего проектирования больших систем [12,13]. Тот факт, что многие инженерные и организационные вопросы также находят отражение в этом подходе, позволяет надеяться, что, избежав участи многих проектов по использованию передовых методик разработки ПО на практике, мы сможем получить действительно полезные практические результаты,.

Описанный проект является частью проводимых международным сообществом исследований подходов к решению проблемы обеспечения развития надежных крупномасштабных программных систем, одного из Больших Вызовов в информатике [1,2]. Тони Хоар (Tony Hoar) предложил вести работы по этой проблеме в двух направлениях: разрабатывать методы и инструменты, в перспективе способные помочь в ее решении, и накапливать примеры использования подобных методов на программных системах реалистичных размеров и сложности (“challenge codes”). Второе направление необходимо как для проверки работоспособности подходов, предлагаемых в качестве возможных решений проблемы, так и для демонстрации путей от исследовательских работ в программной инженерии к их практическим приложениям, для нащупывания области применимости разрабатываемых в академической среде передовых подходов и демонстрации их возможностей инженерам-практикам и представителям промышленности.

Для стимулирования деятельности в этом направлении и вовлечения в него широких кругов разработчиков ПО ИСП РАН передает результаты этого проекта и все инструменты, необходимые для работы с ними, сообществу разработчиков ПО с открытым кодом (open source community). Мы надеемся также на помощь членов этого сообщества в решении более масштабных задач, таких, как формализация большого числа стандартов, входящих в набор Carrier Grade Linux [26], стандартов, регламентирующих встроенные версии Linux и Linux для систем

реального времени, а также стандартов на широко используемые языки программирования.

Благодарности. Мы благодарим Федеральное агентство по науке и инновациям Российской Федерации за предоставленную финансовую поддержку для создания Центра верификации ОС Linux и проведения проекта по формализации LSB.

Литература

[1] http://www.fmnet.info/gc6/

[2] Tony Hoare and Robin Milner, eds. Grand Challenges in Computing. Research. http://www.ukcrc.org.uk/gcresearch.pdf

[3] *ISO 9646. Information Theory — Open System Interconnection — Conformance Testing Methodology and Framework.* ISO, Geneve, 1991.

[4] *ITU-T. Recommendation Z.500. Framework on formal methods in conformance testing.* International Telecommunications Union, Geneve, Switzerland, 1997.

[5] http://www.linuxbase.org/spec

[6] http://www.unix.org/version3/ieee_std.html

[7] G. Bernot. *Testing against Formal Specifications: A Theoretical View.* In Proc. of TAPSOFT'91, Vol. 2. S. Abramsky and T. S. E. Maibaum, eds. LNCS 494, pp. 99–119, Springer-Verlag, 1991.

[8] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. *A formal approach to conformance testing.* In J. de Meer, L. Mackert, and W. Effelsberg, eds. 2-nd Int. Workshop on Protocol Test Systems, pp. 349–363. North-Holland, 1990.

[9] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[10] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. *UniTesK Test Suite Architecture.* In Proc. of FME 2002. LNCS 2391, pp. 77–88, Springer-Verlag, 2002.

[11] V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev, and I. Bourdonov. *Integration of Functional and Timed Testing of Real-time and Concurrent Systems.* In Proc. of PSI 2003, LNCS 2890, pp. 450–461, Springer-Verlag, 2003.

[12] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. *Подход UniTesK к разработке тестов.* Программирование, 29(6):25–43, 2003.

[13] V. Kuliamin. *Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System.* In Proc. of 1-st International Symposium on Leveraging Applications of Formal Methods, Cyprus, October 2004, pp. 311–316.

[14] V. Kuliamin, A. Petrenko, and N. Pakoulin. *Practical Approach to Specification and Conformance Testing of Distributed Network Applications.* In M. Malek, E. Nett, N. Suri, eds. Service Availability. LNCS 3694, pp. 68–83, Springer-Verlag, 2005.

[15] http://www.linuxtesting.ru

[16] http://www.opengroup.org/bookstore/catalog/c610.htm

[17] http://www.caldera.com/developers/devspecs/

[18] *ISO/IEC 9899. Programming Languages — C.* ISO, Geneve, 1999.

[19] ftp://ftp.gnu.org/gnu/glibc

[20] http://ltp.sourceforge.net

[21] http://www.linuxbase.org/download/#test_suites

[22] J. F. Leathrum and K. A. Liburdy. *A Formal Approach to Requirements Based Testing in Open Systems Standards.* In Proc. of 2-d International Conference on Requirements Engineering, 1996, pp. 94–100.

[23] E. Farchi, A. Hartman, and S. S. Pinter. *Using a model-based test generator to test for standard conformance.* IBM Systems Journal, 41:89–110, 2002.

[24] http://www.opengroup.org/testing/testsuites/TestSuiteIndex.html

[25] http://posixtest.sourceforge.net/

[26] http://www.osdl.org/lab_activities/carrier_grade_linux