И.Б.Бурдонов, А.С.Косачев, А.В.Демаков, А.К.Петренко, А.В.Максимов.
Формальные спецификации в технологиях обратной инженерии и верификации программ.
Труды Института системного программирования РАН, N 1, 1999, стр. 35-47.
16 стр.

Формальные спецификации в технологиях обратной инженерии и верификации программ

И.Б. Бурдонов, А.В. Демаков, А.С. Косачев, А.В. Максимов, А.К. Петренко

Аннотация

KVEST (Kernel Verification and Specification Technology) – технология спецификации и верификации программного обеспечения, основанная на автоматизированной генерации тестов из формальных спецификаций. Эта технология была разработана в рамках контракта с Nortel Networks и базируется на опыте, полученном в результате академических исследований. К 1999 году методология и набор инструментов применялись в трех индустриальных проектах верификации телекоммуникационного ПО. Первый проект, The Kernel Verification project, дал название методологии и набору инструментов. Результаты этого проекта присутствуют в Formal Method Europe Application database [28]. Это одно из крупнейших приложений формальных методов, присутствующих в базе данных. Данная статья содержит краткое описание подхода, сравнение со сходными работами и перспективы развития[*].

1. Введение

1.1 Обратная и прямая инженерия ПО

Задачи программной инженерии (software engineering) условно можно разделить на две большие группы – реверс- или обратная инженерия и форвард-инженерия (reverse- and forward-engineering). Разные исследователи и практические разработчики программного обеспечения (ПО) уделяют этим группам разную долю внимания, однако сейчас уже ни одна промышленная разработка не может игнорировать проблемы каждой из этих групп. Форвард-инженерия необходима для того, чтобы поддерживать поступательное развитие ПО, реверс-инженерия необходима для поддержки преемственности функциональности и таких характеристик как надежность, управляемость, открытость к изменениям и др.

В контексте индустриальной разработки и развития ПО важно объединение методов и технологий анализа и создания ПО. При недооценке важности такого объединения легко оказаться в ситуации, когда одни фазы жизненного цикла ПО получают гипертрофированно развитые средства поддержки, что, в частности, приводит к росту объемов ПО, а другие фазы, не имея адекватной поддержки, встречаются с непреодолимыми трудностями. Очевидным примером здесь служит развитие языков программирования, в частности, объектно-ориентированных (ОО) языков и соответствующих компиляторов и интегрированных средств поддержки. Это привело к появлению чрезвычайно громоздких программных комплексов, поддержка, изучение и модификация которых становятся невозможным без специальных методов и инструментов.

В данной статье "реверс-инженерия" часто предшествует "форвард-инженерии". Это объясняется двумя обстоятельствами. Во-первых, авторам ближе этот аспект, поскольку именно с реверс-инженерии начинались работы по созданию технологий, о которых пойдет речь ниже. Во-вторых, и это, может быть, более важно, задачи реверс-инженерии во многих отношениях проще формализуются, и по этой причине на задачах реверс-инженерии имеются реальные предпосылки для апробации самых новых методов и инструментов поддержки разработки и развития ПО. Простота формализации здесь обуславливается тем, что исходный материал для реверс-инженерии – это полностью формализованный программный материал – исходные тексты программ. В случае форвард-инженерии в качестве "исходного материала" выступают существенно менее материальные субстанции: методы проектирования, навыки разработчиков, неформальные спецификации требований и пр. Таким образом, несмотря на то, что целью статьи является развитие концепции единого подхода к решению задач реверс- и

[*] Часть работ по развитию методологии была выполнена в рамках грантов РФФИ 96-0101277 и 99-01-00207.

форвард-инженерии, далее мы будем сохранять тот же порядок рассмотрения аспектов программной инженерии – сначала "реверс", а затем "форвард".

1.2 Формальные методы в разработке ПО

Дать точное определение "формальным методам", как они понимаются в программировании, достаточно затруднительно. Одна из причин этого состоит в том, что программы и методы их компиляции и интерпретации несомненно являются формальными, поэтому и все методы разработки программ легко объявить формальными. Вместе с тем, под термином "формальные методы" скрывается нечто, отличающее рутинное написание текстов на языке программирования от анализа этих текстов и анализа поведения программ, заданных этими текстами, причем анализа по духу близкого к математическим исследованиям, использующего математические нотации и способы рассуждений и доказательств, принятые в математике. В связи с этим многие авторы дают определение "формальных методов" просто как методов разработки программ, в которых используются математическая нотация (notation) и/или математические рассуждения (reasoning). Мы готовы остановиться на этом определении, поскольку не видим причин искать лучшего.

Формальные методы в программировании, по-видимому, появились практически одновременно с самим программированием. Из результатов советской программистской школы наибольшую известность получили работы А.А.Маркова (алгоритмы Маркова) [24] и работы А.А.Ляпунова [25] и его учеников (например, схемы Янова [26]). В более поздние годы много внимания формальным методам в СССР уделялось в работах киевских, новосибирских, ленинградских и московских ученых. Наиболее известной и распространенной формальной нотацией является нотация Бэкуса-Наура, использующаяся для описания синтаксиса формальных языков. Затем можно назвать машину Тьюринга, конечные автоматы (Finite State Machine – FSM or Finite Automata – FA), сети Петри, языки описания взаимодействующих процессов К.А.Хоара (C.A.Hoar) и Р.Милнера (R.Milner) и др.

По естественным причинам практически все работы по формальным методам были нацелены на форвард-приложения. В качестве идеала рассматривалась следующая схема. На языке формальных спецификаций описываются функциональные требования к программной системе. Путем аналитического исследования устанавливается корректность спецификации – спецификация верифицируется. Затем при помощи некоторого инструмента на основе формальных спецификаций генерируется код программной реализации. Несколько более реалистичный сценарий дополнял описанную выше схему процессом постепенного уточнения спецификаций (refining). Каждый шаг уточнения проводится человеком, который направляет процесс уточнения. При этом соответствующие инструменты следят за тем, чтобы очередное уточнение спецификации не пришло в противоречие с исходными спецификациями. В обоих сценариях в качестве итогового результата должна появиться программная реализация, удовлетворяющая всем специфицированным требованиям и не содержащая ошибок.

В 70-е годы появились языки формальных спецификаций, которые с одной стороны имели много общего с языками программирования, а с другой стороны предоставляли специальные средства, сближающие их с математической нотацией и облегчающие рассуждения о свойствах таких формальных текстов. В качестве наиболее известных упомянем CSP [13], CSS [14], VDM [15], SDL[16], LOTOS [17].

Несмотря на это, большая часть исследований по формальным методам по-прежнему сохраняла так называемый "академический" характер. По-видимому, главным исключением служат работы по конечным автоматам (КА), которые нашли самое широкое применение в проектировании и тестировании средств автоматики, связи и вычислительной техники. Опыт использования КА в разработке аппаратуры применялся и в разработке ПО, хотя в существенно меньших масштабах по сравнению с разработкой аппаратуры.

Весьма скромные результаты, продемонстрированные попытками применить формальные методы в реальных проектах, породили распространение скептического взгляда на возможность извлечь пользу из этих методов, соизмеримую с затратами, которые необходимо вложить в дополнительные

работы, связанные с разработкой и анализом формальных спецификаций. Вместе с тем, на отдельных направлениях формальные методы и, в частности, языки формальных спецификаций достигли значимых успехов. Эти успехи, с одной стороны, были обусловлены удачным сочетанием потребностей предметной области и возможностей формальных методов (в первую очередь это проблемы описания телекоммуникационных протоколов; SDL, LOTOS – примеры языков спецификаций, использующихся в этих областях), и с другой стороны, приближением языков спецификации к формам, привычным в традиционном программировании (в первую очередь это Венский метод – Vienna Development Method – VDM и его развитие – языки Z и RAISE).

Еще одним фактором, создавшим предпосылку для продвижения формальных методов в реальное программирование (software production), стал интерес к вопросам реверс-инженерии вообще и к задачам автоматизации тестирования на основе использования формальных спецификаций (тем самым, спустившись с небес на землю, специалисты по формальным методам отбросили мечту об порождении программ без ошибок, а решили использовать свои методы для поиска ошибок, которые неизбежно встречаются в ПО).

Главное преимущество, которое дает использование формальных методов в процессе реверс-инженерии, – это возможность строгого описания интерфейсов и поведения программной системы. Эта возможность, во-первых, позволяет фиксировать знания о функциональности отдельных компонентов и подсистем, знания о правилах взаимодействия, об ограничениях на входные данные, временные характеристики и др. Тем самым, появляется предпосылка для решения самой главной проблемы современной реверс-инженерии. Она состоит в том, что на сегодняшний момент результатом работы по изучению программ (это и есть реверс-инженерия в узком смысле этого слова) является знание отдельного индивида. Это знание не отчуждается от индивида и легко теряется как самим индивидом (и группой, в которой он работает), так и заказчиком реверс-инженерии, как только данный исполнитель переключился на другую работу. Известно, что фирмы производители ПО затрачивают огромные средства на создание документации по ПО. Однако лишь немногие фирмы находят достаточно сил и времени, чтобы поддерживать документацию в актуальном состоянии. Эта ситуация каждый раз порождает необходимость в реверс-инженерии. Реальным выходом из этого бесконечного цикла является фиксация так называемых "программных контрактов" (software contract), которые можно рассматривать как материальное представление знаний о функциональности данного ПО.

Программный контракт описывает синтаксис и семантику интерфейсов систем. Как правило, этот термин используется по отношению к так называемым "интерфейсам прикладных программ" (Application Program Interfaces – API). API – это интерфейс, который предоставляется сущностями, составляющими программу, например, процедурами, функциями, методами ОО классов и т.п.

Помимо собственно фиксации программного контракта, формальная спецификация позволяет систематизировать функциональное тестирование (часто называемое тестированием по методу "черного ящика"). Поскольку формальные спецификации строго описывают требования как на входные данные, так и на ожидаемые результаты, функциональных спецификаций достаточно для того, чтобы провести тестирование внешнего поведения системы.

Заметим, что без строгих спецификаций такой систематизированный подход невозможен, поскольку нет данных ни об области допустимых воздействий на целевую систему, ни о критериях оценки полученных результатов – какие из результатов следует трактовать как правильные, какие как ложные. Это является одной из причин того, что большая часть исследований по тестированию посвящена тестированию на основе исходных текстов. Исходные тексты являются строгим описанием структуры реализации, поэтому они представляются подходящим материалом для извлечения тестов (тестовых воздействий) и для оценки полноты тестового покрытия. Однако, в отличие от функциональных спецификаций, на основании изучения исходных текстов нельзя вынести заключения о критериях проверки соответствия реализации ее функциональным требованиям, в частности, о полноте реализации.

Еще одно обстоятельство является чрезвычайно важным. Если спецификации

формальны, то они могут рассматриваться как "машинно-читаемые". Тем самым появляется предпосылка полностью автоматизировать как генерацию тестов, так и анализ результатов тестирования.

Серьезным направлением в использовании формальных методов в последние десять лет стала “проверка моделей” (model checking). Этот подход демонстрирует компромисс между идеальной мечтой о верификации формальной системы и реальной практикой разработки ПО. Суть похода состоит в построении модели реальной системы и по возможности полной проверке корректности данной модели. Проверка, если возможно, проводится аналитическими методами. Если это невозможно, производится тестирование модели. При этом сложность модели, как правило, выбирается таким образом, чтобы была возможность провести “исчерпывающее” тестирование (exhaustive testing). Слабое место данного подхода – это проблемы построения модели и доказательство того, что модель достаточно содержательна, чтобы на основании модели можно было судить о свойствах реальной системы.

Резюмируя данный краткий обзор позитивных сдвигов в использовании формальных методов в индустриальной разработке ПО, отметим, что сейчас наметилось достаточно четкое разделение методологий и поддерживающих их инструментов, ориентированных на академические исследования и на использование в промышленности ПО. Последние отличаются от первых не только более развитыми средствами поддержки программных проектов, но и средствами, позволяющими перекинуть мостик между спецификациями и собственно целевой системой. К таким средствам в первую очередь относятся компиляторы исполнимых подмножеств языков спецификаций в языки программирования, средства согласования спецификационных и реализационных сущностей, средства, упрощающие конфигурирование целевых и тестовых систем, в которых часть компонентов создается вручную, а часть является результатом генерации из формальных спецификаций. Разнообразие таких возможностей позволяет заключить, что некоторые формальные методологии и наборы их инструментов уже вышли на уровень программных продуктов, и в ближайшее время следует ожидать значительного расширения как их функциональности, так и масштабов их использования.

2. История и основные идеи KVEST

2.1 История разработки и использования KVEST

В 1994 году Nortel Networks (предыдущие названия: Bell-Northern Research, Northern Telecom и Nortel) предложил ИСП РАН разработать методологию и комплект инструментов автоматизации тестирования интерфейсов прикладных программ (Application Program Interface – API). Первым практическим применением методологии стало ядро операционной системы реального времени. Был строго описан программный контракт ядра ОС и созданы тестовые наборы для проверки выполнения реализацией программного контракта.

В случае успеха проекта Nortel получал возможность автоматизированной проверки соответствия программному контракту следующих версий ядра. Кроме того, появлялась возможность улучшить структуру программного продукта в целом, поскольку во время определения программного контракта ИСП предложил установить минимальное и ортогональное множество интерфейсов ядра ОС.

ИСП организовал совместную группу исследователей и разработчиков. Члены группы имели богатый опыт в разработке операционных систем, систем реального времени, компиляторов и в использовании формальных спецификаций для систематического подхода к разработке и тестированию ПО [2, 9, 10].

За первое полугодие ИСП предложил первую версию интерфейсов ядра (Kernel Interface Layer – KIL) и произвел сравнительный анализ доступных методологий специфицирования. KIL был принят с незначительными модификациями. Наиболее подходящим языком спецификации был признан RSL (RAISE Specification Language) [11, 12].

За следующие полгода был разработан предварительный вариант методологии спецификации и генерации тестов, созданы прототипы спецификаций, разработан и реализован генератор тестовых оракулов. Прототип продемонстрировал возможность использования формальных спецификаций в промышленной разработке ПО. В качестве основного типа спецификаций были выбраны имплицитные спецификации. Также были установлены

основные принципы анализа тестового покрытия. KVEST методология использует модификацию СДНФ* критерия для разбиения пространства входных параметров, оценки тестового покрытия и выбора тестовой стратегии.

За следующий год были созданы окончательные версии спецификаций и инструментов для генерации и выполнения тестов. С середины 1996 до начала 1997 года почти все тестовые наборы были получены и ядро ОС успешно протестировано. Результаты тестирования оказались неожиданными для заказчика. Никто не думал, что в критической части ПО, используемого в этой области более десяти лет, будет обнаружено несколько десятков ошибок!

2.2 Основные идеи KVEST

Будем различать методологию и технологию, совокупность технических решений и инструментов, поддерживающих KVEST методологию. (На русском языке слово "методология" звучит слишком претенциозно, тем не менее мы его используем, поскольку это короче, чем "совокупность методов" и полностью адекватно трактовке слова "methodology" в англоязычной литературе). Методология KVEST нацелена на фиксацию программных контрактов и различные способы использования спецификаций программных контрактов.

В качестве основного метода спецификации используется так называемый "моделе-ориентированный" или "ориентированный на состояние" (model based or state based) подход. Этот подход является альтернативным к так называемому "алгебраическому" подходу, синонимами которого являются "аксиоматический" и "действие-ориентированный" (action based) подходы [21, 22]. В моделе-ориентированном подходе спецификация представляет собой привычную программистам совокупность функций и описаний данных, с которыми данные функции оперируют. Как правило, каждой операции (процедуре) целевой системы соответствует некоторая спецификационная функция. Со структурами данных картина существенно сложнее. Различаются "видимые" реализационные данные и скрытые. Видимыми всегда являются входные и выходные параметры операций (процедур, методов). Такие данные, как глобальные переменные, статические области памяти, если они включаются в программный контракт (что, правда, противоречит требованиям грамотного, модульного построения интерфейсов), также объявляются "видимыми". В противном случае они рассматриваются как "скрытые". Видимые данные моделируются в манере близкой к их реализации. Скрытые данные могут моделироваться без прямой привязки к реализации. Это позволяет, во-первых, сделать спецификации в большей степени реализационно-независимыми, во-вторых, более абстрактными, и, следовательно, зачастую более короткими и понятными.

В моделе-ориентированных спецификациях используются два вида описания функций – эксплицитные (явные) и имплицитные (неявные). Первые – знакомы всем пользователям обычных языков программирования. Такие описания представляют собой описание алгоритма вычисления результата функции. Имплицитный способ состоит в описании ограничений на входные параметры (предусловие) и ограничения на совокупность входных и выходных параметров (постусловие). И пред-, и постусловие – это предикат, то есть функция, результатом которой является булевская величина. Предусловие истинно тогда и только тогда, когда входные параметры входят в область допустимых значений данной функции (поведение функции вне области допустимых значений не определено и поэтому не рассматривается и не тестируется). Постусловие истинно тогда и только тогда, когда совокупность значений входных и выходных параметров удовлетворяет требованиям, задающим функциональность, назначение данной функции. Тем самым, постусловие можно рассматривать как формальное определение критерия правильности полученного результата.

Заметим, что имплицитная форма легко позволяет описывать не только конкретное значение, которое должна вычислить функция, но и класс допустимых значений. При описании реальных программных контрактов это очень важно. Рассмотрим два примера. В первом примере не важно, как задавать функцию, в имплицитном или эксплицитном виде. Во втором примере эксплицитная форма практически неприемлема.

Первый пример – это функция, которая выполняет некоторые арифметические вычисления над целыми числами. Известна

* СДНФ – совершенная дизъюнктивная нормальная форма.

формула (одна из возможных), по которой выполняются данные вычисления, обозначим ее f(x), где x – аргументы. В таком случае эксплицитная спецификация целевой функции tf(x) будет иметь вид (X – тип аргументов; Y – тип результатов)

tf : X → Y
tf(x) **is** f(x)

Имплицитная спецификация могла бы иметь следующий вид

post-tf : X × Y → Boolean
post-tf(x, y) **is** y = f(x)

// здесь x – аргумент,
// а y – результат целевой функции

Заметим, что обе рассмотренные спецификации могут быть реализационно-независимыми, поскольку мы не делаем никаких предположений о том, по какой формуле выполняются вычисления в реализации целевой функции. Важно лишь то, что результат целевой функции должен совпадать с результатом вычисления по формуле f(x).

Второй пример: функция, которая выделяет некоторую область памяти, должна возвратить идентификатор этой области. Реализационно-независимая спецификация не может предсказать, какое именно значение примет такой идентификатор. Вместе с тем, в имплицитной форме легко задать ограничение на такой результат. Новый идентификатор не должен совпадать ни с одним из идентификаторов, соответствующих другим областям памяти. Обычно этого требования наряду с требованиями, задающими ограничения типа, достаточно для спецификации такой функции.

Возвращаясь к первому примеру, заметим, что если бы аргументом целевой функции были действительные числа, эксплицитную форму без дополнительных соглашений об интерпретации таких спецификаций использовать было нельзя. При спецификации вычислений над действительными числами приходится дополнительно специфицировать точность вычислений и точность представления результатов. Для этих целей имплицитная форма будет предпочтительнее. Например, спецификация функции tf могла бы иметь следующий вид:

post-tf : X × Y → Boolean
post-tf(x, y) **is** abs(y-f(x)) < delta

Отказ от привязки спецификаций к алгоритмам, используемым в реализации, делает спецификации реализационно-независимыми лишь формально, а не по существу. От хорошей спецификации, в сравнении с реализацией, мы ожидаем более явного, прозрачного описания "смысла". Здесь мы касаемся очень тонких материй. Трудно давать строгие определения, позволяющие сравнить два текста с тем, чтобы указать, в каком из них смысл представлен "более явно". Тем не менее, практически всегда оказывается, что "хорошая" спецификация отличается от реализации более абстрактными структурами данных, используемых в описании интерфейсов. В свою очередь можно сказать, что структура данных "более абстрактна" тогда, когда при ее описании мы в большей степени используем такие математические понятия как множества, отображения, графы, их разновидности и т.п. По-видимому, не имеет смысла говорить о том, что математическая нотация и приемы математических описаний, рассуждений, преобразований лучше соответствующих средств, использующихся программистами в их реальной практике. Важно лишь то, что математическая нотация поощряет аналитика (software designer) взглянуть на программу, ее реализацию, ее поведение с некоторых новых для него позиций. Такое рассмотрение программы в новом ракурсе, по-видимому, и является главной причиной, объясняющей плодотворность использования формальных методов даже там, где полностью автоматическое порождение программ из спецификаций и полная аналитическая верификация программ невозможна.

KVEST в полной мере использует этот прием описания программных контрактов с использованием абстрактных структур данных. Это относится как к типам видимых данных, так и к описанию скрытых данных. Так при описании арифметических функций, работающих с целыми разной длины, KVEST определяет тип целого бесконечной длины. Этот прием позволяет вскрыть немало ошибок в реализации казалось бы несложных функций. В действительности эти функции не сложны только в классической, "бесконечно-значной" арифметике. Там, где появляется арифметика с конечной длиной целого или беззнаковая арифметика, алгоритмы становятся не очевидными. Сопоставление вычислений, выполненных в "бесконечной" и в "конечной" арифметике позволяет единообразным способом провести верификацию библиотек арифметических операций.

В еще большей степени повышение уровня абстракции относится к описанию

моделей скрытых данных. Им соответствуют так называемые "абстрактные" переменные. Структура абстрактных переменных может быть совершенно не похожей на структуру скрытых. Более того, состав абстрактных и скрытых переменных, как правило, существенно различается. Единственное требование, исходя из которого выбирается набор абстрактных переменных, – это возможность моделировать поведение целевой системы, ее функций и, тем самым, как минимум, оценивать получаемые от целевых функций результаты.

2.3 Основные понятия

Рассмотрим некоторую программную систему, содержащую функционально замкнутый набор процедур. Необходимо определить функциональные спецификации внешних интерфейсов, то есть определить программный контракт, а также разработать тестовые наборы, пригодные для проверки выполнения реализацией программного контракта. Поскольку элементами программного контракта являются процедуры, можно говорить о тестировании программного интерфейса, API. Далее будем считать, что программный интерфейс состоит только из процедур. Существуют и другие виды элементов API, такие как операции, функции, методы (в C++), подпрограммы (в Фортране) и т.д. Будем рассматривать все эти термины как синонимы и использовать для них термин «процедура».

Заметим, что речь не идет о тестировании некоторой конкретной реализации программного контракта. Важно построить такую методологию, которая позволяет тестировать поведение программы, не накладывая дополнительных ограничений на внутреннюю структуру реализации. Чтобы подчеркнуть особую важность этого требования, мы называем наши спецификации реализационно независимыми.

Дадим несколько определений. Часть тестовой системы, непосредственно участвующую в процессе выполнения тестов и взаимодействия с тестируемой системой (SUT – system under test), будем называть *тестовым окружением (test harness)*. В некоторых случаях вместо SUT мы будем употреблять термин «целевая система». Основную часть тестового окружения составляют так называемые *тестовые драйверы* (test drivers). Функциональность тестовых драйверов опирается на *систему поддержки времени выполнения* (test bed). Тестовые драйверы обычно разрабатываются с учетом специфики целевой системы, тогда как система поддержки времени выполнения независима от целевой системы.

Мы различаем два уровня тестовых драйверов. Назовем базовым драйвером тестовый драйвер для некоторой процедуры, который выполняет следующие задачи:

- проверяет выполнение предусловия целевой процедуры на некотором наборе входных параметров;
- вызывает целевую процедуру с данными входными параметрами и сохраняет полученные значения выходных параметров;
- выносит вердикт о корректности работы целевой процедуры;
- собирает информацию, необходимую для оценки тестового покрытия, или исследует причины ошибки.

Назовем *скрипт-драйвером* тестовый драйвер для некоторой целевой процедуры или группы процедур, который выполняет следующие задачи:

- читает значения параметров тестирования;
- генерирует множество входных параметров в соответствии с полученными параметрами тестирования;
- вызывает базовый драйвер с некоторым набором входных параметров;
- если необходимо, производит дополнительную проверку корректности работы целевой процедуры и выносит вердикт;
- если желаемое тестовое покрытие не достигнуто, продолжает генерировать наборы значений входных параметров и вызывать базовые драйверы.

Назовем *тестовым планом* программу, которая определяет порядок вызовов скрипт-драйверов с данными параметрами тестирования, проверяет условия вызова и корректность завершения работы скрипт-драйверов.

Кроме базовых и скрипт-драйверов и интерпретатора тест-планов тестовое окружение также содержит репозиторий и инструменты для выполнения тест-планов и анализа результатов, хранящихся в репозитории. Репозиторий содержит информацию о всех прогонах тестов, достигнутом тестовом покрытии для различных процедур с различными критериями тестового покрытия и информацию обо всех ситуациях, когда тестовый драйвер вынес вердикт о несоответствии спецификации и

реализации.

2.4 Шаги методологии

KIL методология состоят из нескольких шагов (см. рис. 1):

- определение состава программного контракта;
- разработка спецификаций;
- генерация тестовых наборов;
- выполнение тестов и анализ результатов.

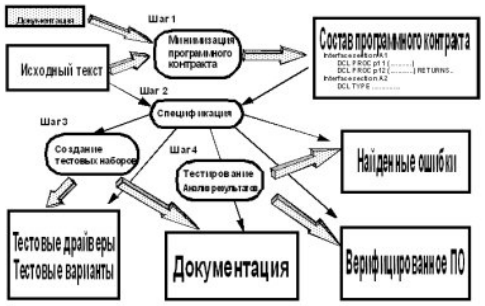

Рис. 1. KVEST методология – шаги и результаты

2.4.1 Определение состава программного контракта

Цели этого шага:

- определить минимальный и ортогональный интерфейс;
- скрыть внутренние структуры данных и детали реализации.

Следуя этим целям, мы должны минимизировать ограничения на возможные реализационные решения и знания, необходимые для использования программы, и сделать возможным разработку долгоживущих тестовых наборов для проверки выполнения программного контракта.

2.4.2 Разработка спецификаций

Цели:

- Строго описать функциональность;
- Создать входную информацию для генерации тестов.

Базовые драйверы могут быть сгенерированы полностью автоматически.

2.4.3 Генерация тестовых наборов

Цели:

- сгенерировать тестовые наборы;
- дополнить сгенерированные тестовые наборы компонентами, разработанными вручную компонентами (MDC – manually-developed component).

Большинство MDC представляют собой конвертеры между модельным и реализационным представлением данных, инициализаторы тестовых структур данных и итераторы. Тестовые наборы генерируются на основе спецификаций и MDC. После завершения генерации дополнительная модификация тестовых наборов не требуется.

Рис. 2. Общая схема KVEST технологии

2.4.4 Выполнение тестов и анализ результатов

К инструментам для пропуска тестов и анализа результатов предъявляются следующие требования:

- автоматизация выполнения тестов;
- сбор трассировочной информации и вычисление достигнутого тестового покрытия;
- предоставление навигационных возможностей;
- возможность «инкрементального тестирования», то есть восстановление целевой системы после ошибки или краха и продолжение выполнения тестов с места, где оно было прервано.

На рис. 2 обобщенно представлены исходные данные и результаты, которые получаются при использовании KVEST методологии.

2.4.5 Метод тестирования

При разработке тестового драйвера необходимо решить три проблемы:

- как сгенерировать оракула, то есть программу, которая выносит вердикт о корректности работы целевой процедуры;
- как оценить полноту тестового покрытия;
- как перебирать комбинации тестовых входных данных.

Тестовые оракулы очень похожи на постусловия. Обе функции возвращают логическое значение, имеют те же самые параметры и возвращают истинное значение в том и только в том случае, когда целевая процедура производит корректный результат. Таким образом, генерация оракулов значительно проще, если имеются постусловия.

Критерий тестового покрытия – это метрика, определенная в терминах

реализации или спецификации. Наиболее известными критериями покрытия в терминах реализации являются:

- C1 – все операторы покрыты;
- C2 – все ветви покрыты.

В случае использования спецификаций для определения критерия покрытия используется так называемый подход тестирования доменов, при котором пространство входных значений разбивается на области. Каждая область соответствует классу эквивалентности. Разбиение может быть выведено из спецификаций, которые описывают ограничения на входные параметры и свойства выходных параметров целевой процедуры, которые явно присутствуют в пред- и постусловиях формальных имплицитных спецификаций. Таким образом, исходя из имплицитных спецификаций, мы можем успешно решить проблему оценки тестового покрытия.

Хорошее покрытие доменов, даже дополненных интересными точками (например, находящимися на границах доменов), не гарантирует хорошего покрытия реализационного кода. Тем не менее, наш опыт показывает, что средний уровень покрытия при KVEST технологии составляет от 70 до 100 процентов операторов реализации.

Мы различаем два уровня критериев покрытия. Первый уровень – это покрытие всех ветвей постусловия. Второй – покрытие всех дизъюнктов (элементарных конъюнкций) в постусловии, представленном как СДНФ, также принимая во внимание предусловие. KVEST технология позволяет производить разбиение в терминах ветвей и СДНФ спецификации полностью автоматически. Одна из наиболее сложных проблем – вычисление достижимых дизъюнктов и удаление недостижимых дизъюнктов. В KVEST эта проблема решается путем использования специальных приемов написания предусловий.

Наблюдение за достигнутым тестовым покрытием производится скрипт-драйверами. Основываясь на этих данных, скрипт-драйвер может подстраивать параметры тестирования и/или длительность тестирования.

2.5 Технология генерации тестов

2.5.1 Классификация API

Для начала рассмотрим классификацию API. Классификация определяет выбор способа генерации тестов, применимого к данной процедуре или группе процедур.

Мы рассматриваем пять основных классов API и некоторые расширения этих классов, включающие тестирование параллельного выполнения процедур и тестирование процедур, которые могут вызвать аварийное завершение программы.

Эти классы упорядочены – первый класс накладывает самые строгие ограничения на процедуру, а последующие классы постепенно эти ограничения ослабляют:

Класс 1. Входные данные могут быть представлены в литеральной (текстуальной) форме, взаимозависимости между параметрами отсутствуют. Такие процедуры могут тестироваться поодиночке, так как для генерации значений входных параметров и анализа результатов не требуются никакие другие целевые процедуры.

Примеры взаимозависимостей между параметрами будут приведены ниже.

Класс 2. Взаимозависимости между входными параметрами отсутствуют. Однако значения входных параметров не обязательно имеют литеральный вид. Такие процедуры также могут тестироваться поодиночке.

Пример: Процедура с указателем в качестве входного параметра.

Класс 3. Существуют некоторые взаимозависимости между входными параметрами, однако возможно тестирование отдельной процедуры.

Пример: Процедура с двумя параметрами – массив и значение одного из элементов массива.

Класс 4. Процедуры не могут тестироваться по отдельности, поскольку значения некоторых входных параметров могут быть получены только в результате вызова других процедур группы и/или результат работы может быть проанализирован только вызовом других процедур.

Пример: Процедуры, реализующие операции со стеком, которые получают стек в качестве параметра.

Класс 5. Процедуры не могут тестироваться по отдельности. Часть входных и выходных данных скрыта, то есть пользователь не имеет к ним прямого доступа.

Пример: объекты классов с закрытым внутренним состоянием, группы процедур, имеющие доступ к переменным, не видимым для пользователя.

Расширение классов API для случая параллельного выполнения. Теоретически, процедуры всех классов должны тестироваться параллельно на случай, когда

между ними имеется какое-либо взаимодействие. На самом деле, имеет смысл тестировать параллельно только процедуры пятого класса, поскольку они разделяют общие ресурсы и ошибки наиболее вероятны в этом случае.

Пример: Процедуры, работающие с почтовыми ящиками (прием и передача сообщений и т.п.)

Расширение классов API для случая процедур, которые могут вызвать аварийное завершение программы. Существует специальный класс процедур, для которых аварийное завершение программы является корректной реакцией в некоторых случаях.

Пример: Процедура, в которой может произойти деление на нуль, полученный в качестве входного параметра. Если эта процедура не возвращает никакого кода возврата, то нормальной реакцией на ошибку может быть завершение программы.

2.5.2 Схема скрипт-драйвера. Пример API класса 5

Вышеприведенная таксономия является хорошей основой для классификации способов генерации тестов. Для API класса 1 возможна полностью автоматическая генерация тестов. Все остальные классы требуют некоторых дополнительных усилий по созданию MDC. Эти усилия плавно возрастают от класса 2 к классу 5. Специальные расширения классов требуют больших усилий, чем сами классы.

Усилия по созданию MDC обусловлены сложностью написания и отладки скрипт-драйверов. Ниже мы рассматриваем только одну схему скрипт-драйвера – для API класса 5. Все скрипт-драйверы имеют похожую структуру. Основное различие в пропорции между объемом автоматически сгенерированных и созданных вручную компонент. Скрипт-драйверы класса 1 генерируются полностью автоматически, класса 2 – почти автоматически и т.д.

Скрипт-драйвер – это программа, которая составляется и компилируется по KVEST технологии. Общая схема скрипт-драйвера определена формальным описанием, которое называется скелетоном. Для каждого класса API имеется свой скелетон. Каждый скрипт-драйвер состоит из деклараций и тела. Декларации генерируются автоматически на основании списка тестируемых процедур и их спецификаций.

Тело скрипт-драйвера начинается с разбора параметров тестирования. Эти параметры определяют глубину тестирования, то есть уровень критерия тестового покрытия и некоторые специфичные данные, такие как интервалы значений, продолжительности тестирования и т.п.

До начала тестирования производится инициализация. Например, до начала тестирования процедур записи/чтения файла необходимо этот файл открыть. Такая инициализация пишется вручную. После инициализации начинает работу основная часть скрипт-драйвера.

Скрипт-драйвер класса 5 реализует общий алгоритм обхода абстрактного конечного автомата (FSM – Finite State Machine). Цель алгоритма – обойти все состояния автомата и все переходы между состояниями. Состояния конечного автомата соответствуют классам состояний модели исходной подсистемы. Каждый переход соответствует вызову тестируемой процедуры.

Алгоритм скрипт-драйвера зависит только от модели исходной подсистемы, не используя никаких деталей реализации, которых нет в спецификации.

Наиболее интересный аспект алгоритма скрипт-драйвера состоит в отсутствии явного описания конечного автомата. Прямое описание конечного автомата требует дополнительных усилий, которых по KVEST технологии можно избежать. Существуют попытки построения конечного автомата по имплицитным спецификациям [7]. Однако пока никто не смог предложить полностью автоматического способа такого построения.

Вместо явного описания конечного автомата, KVEST использует его косвенное представление. Для описания конечного автомата создатель скрипт-драйвера должен иметь мысленную модель конечного автомата и задать функцию, вычисляющую состояние конечного автомата на основании модельного состояния подсистемы.

Рассмотрим более детально алгоритм скрипт-драйвера класса 5. Для примера рассмотрим тестирование группы процедур. Предположим, что мы прошли несколько состояний конечного автомата, то есть несколько раз вызвали целевые процедуры. Теперь мы должны определить следующий переход. Элементарный цикл тестирования состоит из следующих шагов:

- Выбираем очередную процедуру из группы.

- Вызываем итераторы, которые формируют набор значений входных параметров для этой процедуры.
- Если итераторам удалось сформировать новый корректный набор значений, удовлетворяющий предусловию, скрипт-драйвер вызывает соответствующий базовый драйвер с этими параметрами.
- Если корректного набора значений сформировать не удалось, возвращаемся к началу и повторяем попытку для следующей процедуры.
- После того, как базовый драйвер закончил работу, скрипт-драйвер проверяет вынесенный им вердикт.
- Если вердикт положительный (элементарный шаг тестирования закончился успешно), скрипт-драйвер вызывает функцию вычисления очередного состояния конечного автомата, сбрасывает трассировочную информацию об очередном состоянии и переходе и продолжает обходить конечный автомат.

2.5.3 Композиция тестового набора

Вернемся к вопросу соединения MDC и автоматически генерируемых компонент. Скрипт-драйверы создаются следуя требованиям соответствующего скелетона. Нам необходимо 5 скелетонов для последовательного тестирования API классов 1-5; один скелетон для параллельного тестирования и пять скелетонов для тестирования процедур, которые могут вызвать аварийное завершение программы. На основе скелетона и спецификаций целевых процедур по KVEST технологии генерируется шаблон скрипт-драйвера. Для класса 1 шаблон представляет собой готовую программу. В шаблонах остальных классов имеются гнезда, заполненные значениями по умолчанию для инициализаторов и итераторов. Если разработчик скрипт-драйвера не нуждается в улучшении содержимого гнезд, шаблон может быть скомпилирован и выполнен. Эта обычная ситуация для класса 2. Для остальных классов разработчик обычно все же должен добавить некоторые итераторы и инициализаторы. В любом случае, для классов 4-5 он должен определить функцию, возвращающую текущее состояние конечного автомата.

Базовые драйверы, вызываемые скрипт-драйверами, генерируются полностью автоматически. Единственные MDC, вызываемые из базовых драйверов – это конвертеры данных из модельного представления в реализационное и наоборот. Модельное представление данных отличается от реализационного уровнем абстракции. Например, модели могут использовать «бесконечные» представления целых чисел, множеств, отображений и других структур данных, подходящих для спецификации. Иногда модельное представление очень похоже на реализационное. В этом случае такая трансформация производится по стандартному алгоритму преобразования языка спецификации в язык программирования.

Рис. 3. Схема генерации тестового набора, независимого от целевого языка программирования

KVEST использует генераторы тестов, независимые от языка реализации целевой системы. У всех генераторов на входе и выходе текст на языке RSL. Единственные компоненты, которые пишутся на языке реализации – это конвертеры данных. Эти компоненты находятся вне зоны ответственности генераторов тестов. Таким образом, результатом процесса генерации тестов является полный текст тестового набора на языке RSL, который затем транслируется в язык реализации. Чтобы перенести тестовый набор, построенный по KVEST технологии, с одного целевого языка в другой, пользователю необходимо переписать все конвертеры данных и предоставить транслятор с RSL в целевой язык, а также СПВВ. Схема генерации тестов находится на рис. 3.

2.5 Интеграция обратной и прямой инженерии ПО

В 1999 году KVEST начинает применяться для совместной работы проектировщиков и верификаторов. Группа разработчиков в Оттаве будет разрабатывать проектную документацию и реализацию, в то время как другая группа в Москве параллельно будет разрабатывать формальные спецификации и тестовые наборы. Такая схема позволяет улучшить

качество проектной документации и создать тестовые наборы еще до того, как будет готова реализация. Это один из способов использования KVEST в процессе прямой инженерии ПО.

В 1998 году ИСП начал исследовательские работы по генерации документации на естественном языке. Результатом этой работы стал прототип, который демонстрировался на конференции ZUM'98 (Берлин, сентябрь 1998)

Этот прототип на основе формальных и неформальных компонентов спецификации синтезирует документацию в стиле UNIX программы man. Результаты демонстрируют реальную возможность генерации документации. Актуальность документации проверяется тестами, которые генерируются из того же источника информации – спецификаций на языке RSL. Работа по расширению генерируемых форм документации и улучшению качества языка продолжается.

3. Современное состояние методов генерации тестов из формальных спецификаций

В этом разделе мы будем рассматривать системы, в которых, с одной стороны, в процесс верификации используются формальные спецификации, а с другой предлагается достаточно общая технологическая схема, поскольку реализация теоретического решения отдельных задач сталкивается со значительными трудностями при практическом их применении в процессе верификации индустриального ПО.

3.1 Система ITEX (Interactive TTCN Editor and eXecutor)

ITEX [29] – это система для разработки тестов для систем коммуникации. Она включает инструменты TTCN и ASN.1 для анализа и проектирования, тестовый эмулятор и поддержку для генерации полных выполнимых тестовых наборов (BTH). Основные возможности ITEX:

- Тестовый набор состоит из наборов тестовых параметров, заданных в форме таблиц;
- ITEX предоставляет набор хорошо интегрированных инструментов для создания и поддержки Абстрактных Тестовых Наборов (ATH), написанных в TTCN;
- IETX поддерживает такие фазы разработки тестовых наборов как генерацию наборов тестовых параметров, редактирование, верификацию и выполнение.

Этот набор инструментов хорошо интегрирован с SDT – системой проектирования SDL спецификаций. Тестовые наборы, описанные с помощью TTCN, могут быть преобразованы в форму, которая позволяет тестировать и реализацию на языке программирования и SDL спецификации.

Основным недостатком данного подхода в контексте наших исследований является невозможность тестирования API. TTCN не позволяет использовать указатели и другие программные сущности, которые не имеют литерального представления.

Кроме того, очень серьезным ограничением SDL-подобных спецификаций является их эксплицитность. Это значит, что довольно легко построить модели и прототипы на основе этих спецификаций, но очень трудно разработать систему ограничений, которая определяет класс возможных реализаций. Имплицитные спецификации решают эту проблему.

3.2 ADL/ADL2

Этот подход [27] наиболее похож на работу нашей группы. Из формальных спецификаций ADL генерирует тестовые оракулы и скелетоны для построения тестовых драйверов и документации. Не очень большое, но интересное отличие состоит в том, что ADL использует не распространенный универсальный язык спецификации, а расширение языков C и C++. В рамках KV проекта на этапе создания прототипа был разработан язык SPP – аналогичное расширение целевого языка. Он документирован в KV Project Report [20]. Аналогичный способ расширения был предложен Барбарой Лисков (B.Liskov) [6]. Существуют идеи расширения Java и других объектно-ориентированных языков, направленные на разработку ПО по принципу «дизайн по контракту» (design-by-contract) [5, 27, 31]. Однако, несмотря на очевидные преимущества лучшего приема таких языков сообществом разработчиков ПО, до общей концепции еще далеко, общая нотация не выработана.

Разницу в результатах KVEST и ADL можно объяснить разницей в классах API, для которых та и другая методологии могут предоставить средства спецификации и генерации тестов. ADL предоставляет инструменты для автоматизации генерации тестов только для процедур, которые могут тестироваться независимо, с параметрами,

допускающими независимый перебор. По KVEST классификации это процедуры первого и второго классов. Это значит, что процедуры с зависимыми параметрами, процедуры, которые требуют совместного тестирования, например open/close, или процедуры, которые необходимо тестировать параллельно, например, lock/unlock, или send/receive, отбрасываются. Кроме того, ADL не распознает API класса 1, для которых возможна автоматическая генерация всего тестового набора, включая наборы тестовых параметров и тестовые оракулы.

Интересным моментом ADL является возможность генерации документации на естественном языке. Важно, что один и тот же механизм используется для документирования как целевой системы, так и тестовых наборов. Похоже, что авторы ADL сознательно не использовали технологий из области NLG (Natural Language Generation). Это понятно из практических соображений, но не значит, что современные методы генерации текста на естественном языке не могут помочь в генерации программной документации. Возможности KVEST по генерации документации реализованы в прототипной форме. Однако, в противоположность ADL KVEST использует компьютерную грамматику и словарь английского языка для анализа и генерации фрагментов на естественном языке. Это способствует уменьшению числа ошибок в тексте на естественном языке и делает текст более читабельным без дополнительной ручной правки.

Значительным преимуществом ADL2 в сравнении с KVEST является возможность спецификации и тестирования классов объектно-ориентированных языков программирования. Этот недостаток KVEST объясняется ограничениями языка спецификации RSL. Расширение KVEST для верификации объектно-ориентированного программного обеспечения планируется в 1999 году.

3.3 Использование тестовых оракулов, сгенерированных из программной документации

Работа [8] является исследовательской и не может рассматриваться как технология, пригодная для промышленного использования. Основной интерес в этом исследовании представляет анализ факторов, которые, по мнению авторов, препятствуют широкому распространению формальных спецификаций для индустриального тестирования программного обеспечения. Авторы формулируют пять основных проблем, общее решение которых, по их мнению, при существующем положении дел невозможно. Эти пять проблем имеют много общего с набором характеристик, на которых базировалась классификация API в KVEST. Таким образом, Д.Петерс (D. Peters) и Д.Парнас (D.Parnas) и мы пришли к общему пониманию, что это ключевые проблемы в задаче автоматизации тестирования, использующего формальные спецификации. KVEST продолжает исследования в этом направлении и предлагает технологическую схему для частичной автоматизации разработки тестовых наборов для всех классов API.

3.4 Формальный вывод конечного автомата для тестирования класса

Эта работа [7] также является исследовательской. В то же время, она представляет интерес, поскольку предлагает схему тестирования группы процедур, аналогично схеме, использующейся в KVEST. В качестве языка спецификации используется Object-Z, а в качестве целевого языка программирования – C++. Задача формулируется следующим образом: построить тестовые наборы для проверки соответствия реализации и спецификации, используя формальные спецификации методов класса. Как критерий тестового покрытия используется объединение двух критериев: покрытие всех классов эквивалентности, которые представляют собой области, полученные в результате анализа разбиений и, затем, проверка результатов на границах и рядом с границами.

Авторы этой работы не пытаются решить проблему полностью автоматической генерации тестов. Также они не делают попыток поддержки каких-либо элементов подготовительной фазы с помощью каких-либо инструментов. Однако, все эти шаги описаны очень систематически и могут быть сведены к различным преобразованиям спецификаций.

Анализ разбиений и границ производится вручную в соответствии с предложенной авторами методологией. Аналогичным образом строятся спецификации оракулов. Оракул, скомпилированный в C++, вызывает целевую процедуру и проверяет соответствие результатов ее работы спецификации.

Наиболее интересна сама схема тестирования, в соответствии с которой динамически генерируются тестовые последовательности вызовов целевых процедур. Генерация контролируется описанием конечного автомата, который представляет абстрактный граф переходов между состояниями тестируемого класса. Авторы описывают методологию построения спецификаций для классов состояний и переходов между ними, одновременно рассматривая проблему исключения недостижимых состояний.

Теоретическая слабость этого подхода состоит в отсутствии попыток создания формальной методологии создания спецификации переходов. Очевидно, что при попытках применить этот метод к задачам реальной сложности обнаружатся серьезные проблемы. Понятно, что вывод тестов из спецификаций производится в основном вручную, что ограничивает применимость этого метода в индустрии ПО.

Основное отличие KVEST от этой работы состоит в том, что KVEST не требует полного описания конечного автомата, который моделирует состояние целевой системы. Вместо этого KVEST предлагает универсальный алгоритм, который динамически поддерживает доступные состояния и переходы между парами состояний.

4. Заключение и направления дальнейшей работы

Опыт KVEST показал, что формальные методы могут использоваться в промышленной разработке ПО. Уровень сложности и размеры приложений KVEST поддерживают этот тезис.

Вместе с тем, текущее состояние KVEST и состояние дел в целом диктуют необходимость интенсивного развития подходов, методологий и поддерживающих их инструментов. Можно сформулировать следующие важные проблемы, требующие своего решения:

- Пользователи языков программирования не владеют языками спецификаций и формальными методами, это является главным фактором, сдерживающим более широкое использование формальных методов;
- Методологии, технологии, CASE системы, поддерживающие разработку ПО, как правило, нацелены на разработку и анализ структур (архитектур) реализаций. Этот подход затрудняет рассмотрение собственно функциональности ПО. Методологий, которые удачно сочетают преимущества обоих подходов: структурного и функционального, – пока нет;
- Статические (чисто формальные) методы анализа программ предоставляют исчерпывающие решения проблем, но применимы лишь для фрагментов реальных систем. Динамические методы типа моделирования или тестирования опираются на некоторые эвристики, поэтому не могут служить базой для исчерпывающего анализа. В связи с этим встает проблема интеграции статических и динамических методов, с тем чтобы извлечь преимущества каждого из них.

Для решения этих проблем мы видим следующие пути:

- **Сближение языков спецификации и языков программирования.**

Исходя из того, что заставить программиста изучать не только новый для него язык спецификации, но и просто новый язык программирования невозможно, легко прийти к выводу, что надо сделать переход от языка программирования к языку спецификации более незаметным. Попытки сблизить эти языки делались уже давно. Примерами таких языков служат расширение CLU [18], Alphard [19], Eiffel[27, 31], iContract [5], SDL [16], SPP [20].

Некоторые языки, например, Larch [23], предлагают делать это сближение за счет упрощения собственно спецификационной части при одновременном расширении средств для отображения формальной модели в язык реализации. Другие языки, например SDL, пользуясь спецификой проблемной области, заимствуют возможности языков программирования и за счет этого позволяют генерировать исполнимый код прямо из спецификаций.

Имеется и вполне удачный опыт встречного движения. Например, авторы ADL путем небольших добавлений расширяют C, C++, Java, и IDL. В результате для пользователя языка программирования нет никаких проблем, по крайней мере, в чтении спецификаций.

KVEST в своем перспективном развитии рассматривает как основной последний из перечисленных подходов. В настоящее время разрабатывается версия системы, которая в качестве инструмента специфицирования предлагает средства C++, возможно, пополненные несложным в

реализации и в понимании “синтаксическим сахаром”.

Заметим, что концепция такого пополнения нуждается в глубокой проработке. Последствия “простых решений” легко видеть на примере ADL. Отсутствие методов и соответствующих средств для повышения уровня абстракции не позволяют пользователям ADL создавать спецификации повышенного уровня абстракции и в достаточной степени автоматизировать генерацию тестов для групп процедур или классов.

- **Синтез методов анализа и разработки программ, направленных на описание функциональности и на описание структуры реализации.**

Некоторые аспекты поведения ПО плохо укладываются в рамки рассмотрения API. В качестве примера можно привести стеки телекоммуникационных протоколов и распределенные системы. Такого рода ПО хорошо описывается в форме исполнимых моделей, построенных на основе конечных автоматов, сетей Петри, специальных видов автоматов, например, типа CCS [14]. Общим недостатком таких подходов являются трудности с заданием инвариантов, охватывающие распределенные события. В частности, эта проблема проявилась при попытке обнаружения потенциальных нежелательных взаимовлияний (Feature Interaction). Одним из путей интеграции структурных и функциональных методов являются КА, расширенные описанием ограничений на переходы [4]. Этот подход близок к технике описания и тестирования групп процедур, предложенный в KVEST. Тем самым, KVEST может быть расширен соответствующими средствами для описания протоколов, распределенных систем и других видов ПО, требующих сочетания структурных и функциональных спецификаций.

- **Интеграция статических и динамических методов в форвард- и реверс-инженерии.**

Из потенциально возможных приложений синтеза статических и динамических методов разработки/анализа выделим средства для управления уровнем абстракции и генераторы моделей для статического и/или динамического анализа поведения программ.

В качестве средств управления уровнем абстракции в KVEST разрабатывается методическая и инструментальная поддержка для “подъема” и “спуска” (“upwarding” and “downwarding”) и средства трансформации эксплицитных спецификаций в имплицитные (в форме пост-условий). Вплоть до последнего времени эти работы выполнялись полностью вручную. Сейчас разрабатываются некоторые из запроектированных “транформаторов”.

Модели, о которых идет речь, являются обобщенным описанием сценариев использования целевой системы. Такие сценарии могут использоваться как для статического анализа (в этом и состоит идея “model checking”), так и для генерации тестовых последовательностей. В KVEST роль таких моделей играли КА в 5-ом виде тестовых драйверов. То, что такой КА разрабатывается вручную, помимо проблем связанных со стоимостью, сроками его разработки, ставит вопрос о соответствии между КА и спецификациями процедур, об отношении критериев тестового покрытия, определенных на основе спецификаций и на основе собственно КА. В настоящее время имеются работы по частичной автоматизации построения такого КА. К сожалению, в них не рассматриваются вопросы, связанные с извлечением КА из спецификаций функций с побочным эффектом. В KVEST делается попытка решить эту задачу на основе использования спецификаций скрытых (абстрактных) состояний целевых систем.

Литература

1. B.Algayres, Y.Lejeune, G.Hugonnet, and F.Hantz. *The AVALON project: A VALidation Environment For SDL/ MSC Descriptions.* // In: 6th SDL Forum, Darmstadt, 1993.
2. I.Burdonov, V.Ivannikov, A.Kossatchev, G.Kopytov, S.Kuznetsov. *The CLOS Project: Towards an Object-Oriented Environment for Application Development.* – In Next Generation Information System Technology, Lecture Notes in Computer Science, volume 504, Springer Verlag, 1991, pp. 422-427.
3. I.Burdonov, A.Kossatchev, A.Petrenko, S.Cheng, H.Wong. *Formal Specification and Verification of SOS Kernel.* // BNR/NORTEL Design Forum, June 1996.
4. Pansy Au and Joanne M. Atlee. *Evaluation of a State-Based Model of Feature Interactions.* // Proceedings of the Fourth International Workshop on Feature Interactions in Telecommunications Software Systems, June 1997, pp. 153-167.
5. R.Kramer. *iContract – The Java Design by Contract Tool.* // 4th conference on OO technology and systems (COOTS), 1998.
6. B.Liskov, J.Guttag. *Abstraction and Specification in Program Development*. – The MIT Press, McGraw-Hill Book Company, 1986.
7. L.Murray, D.Carrington, I.MacColl, J.McDonald, P.Strooper. *Formal Derivation of*

Finite State Machines for Class Testing. // Lecture Notes in Computer Science, volume 1493, pp. 42-59.
8. D.Peters, D.Parnas. *Using Test Oracles Generated from Program Documentation*. // IEEE Transactions on Software Engineering, 1998, Vol. 24, N. 3, pp.161-173.
9. A.K.Petrenko. *Test specification based on trace description*. // Software and Programming, New York (translated from Programmirovanie), No. 1, Jan-Feb. 1992, pp. 26-31.
10. A.K.Petrenko. *Methods of debugging and monitoring of parallel programs.* // Software and Programming, N. 3, 1994.
11. The RAISE Language Group. *The RAISE Specification Language*. – Prentice Hall Europe, 1992.
12. The RAISE Language Group. *The RAISE Development Method*. – Prentice Hall Europe, 1995.
13. A.R.Hoare. *Communicating Sequential Processes*. – Prentice Hall, 1985.
14. R.Milner. *Communication and Concurrency*. – Prentice Hall, 1989.
15. D. Bjorner et al eds. *The Vienna Development Method: The Meta-Language*. – Lecture Notes in Computer Science, volume 61, Springer Verlag, 1978.
16. *Specification and Design Language*. ITU-T recommendation Z100.
17. P.H.J. van Eijk et al eds. *The Formal Description Technique LOTOS*. – North Holland, 1989.
18. Barbara Liskov et al. *CLU Reference Manual*. – Lecture Notes in Computer Science, volume 114, Springer Verlag, 1981.
19. Mary Shaw. *Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators*. // Communications of the ACM, Vol. 20, N. 8 (August 1977), pp. 553-563.
20. *SPP – specification language description.* – KV project report, Nortel (Northern Telecom), May 1995.
21. C.A.R.Hoare. *An axiomatic basis for programming.* // Communications of the ACM, Vol. 12, N. 10 (October 1969), pp. 576-583.
22. C.A.R.Hoare. *Proof of correctness of data representations*. // Acta Informatica, 1(4): 271-281, 1972.
23. J. Guttag et al. *The Larch Family of Specification Languages.* // IEEE Software, Vol. 2, N. 5, pp. 24-36 (September 1985).
24. А.А.Марков. Теория алгорифмов. // Москва, Изд-во АН СССР, 1954.
25. А.А.Ляпунов. О Логических схемах программ. Проблемы кибернетики, Москва, вып. 1, 1958.
26. Ю.И.Янов. О логических схемах алгоритмов. Проблемы кибернетики, вып. 1, Москва, 1958.

Ресурсы сети Internet

27. http://www.eiffel.com/doc/manuals/language/intro/
28. http://www.fme-nl.org/fmadb088.html
29. http://www.kvatro.no/products/itex/itex.htm
30. http://www.sun.com/960201/cover/language.html
31. http://www.elj.com/eiffel/intro/