От игровых сценариев к коду – достижимая мечта[1]

Давид Харел (David Harel), Научный институт Вейцманна (Weizmann)

Схема разработки для сложных реактивных систем ведёт от метода фиксации дружественных пользователю требований, называемых игровыми сценариями, до полных описаний поведения частей системы, а от них – к конечной реализации.

В статье 1992 года в журнале *«Компьютер»*[1] я пытался представить оптимистичный взгляд на будущее методов разработки сложных систем. Исследования, проведённые с того момента, только подтверждают этот оптимизм, что я и попытаюсь показать.

Эта статья представляет общую, довольно поверхностную схему разработки, объединяющую давно известные идеи с более новыми. Эта схема позволяет перейти от метода фиксации высокоуровневых, дружественных пользователю требований, которые я называю игровыми сценариями, посредством богатого языка для описания последовательности сообщений, к полной модели системы, и от неё – к конечной реализации.

Циклический процесс проверки системы на соответствие требованиям и синтеза её частей по требованиям является центральным в этом вопросе. Статья уделяет особое внимание языкам, методам и автоматизированным инструментам, которые позволяют осуществлять плавные, но точные переходы между различными этапами схемы. В противоположность системам, использующим базы данных, эта статья сосредотачивает внимание на системах, которые обладают преимущественно реагирующими особенностями, управляемыми событиями. Для этих систем моделирование и поведенческий анализ являются наиболее критическими и проблемными моментами.

МОДЕЛИРОВАНИЕ СИСТЕМЫ

На протяжении многих лет главными подходами к высокоуровневому моделированию систем были структурный анализ и структурное проектирование (СА/СП) и объектно-ориентированный анализ и проектирование (ООАП). Эти два подхода к моделированию отстоят примерно на десятилетие по начальной концепции и эволюции. На протяжении многих лет оба подхода породили визуальные формализмы для фиксации различных частей модели системы, особенно её структуры и поведения. Связь структуры и поведения является решающей и отнюдь не простой. В методологии СА/СП, например, каждая функция или действие системы связаны с конечным автоматом или диаграммой состояний[2], которые описывают их поведение. В методологии ООАП, как, очевидно, и в унифицированном языке моделирования (УЯМ)[3] и его исполнимой основе ИУЯМ[4], каждый класс связан с диаграммой состояний, которая описывает поведение каждого рассматриваемого объекта. Приложения «Структурный анализ и структурное проектирование» и «Объектно-ориентированный анализ и проектирование» дают некоторые начальные сведения по этим подходам к моделированию.

Необходимой частью любого серьёзного подхода к моделированию является строгая семантическая основа для конструируемой модели, особенно для поведенческих частей модели и их связи со структурой. Это та семантика, которая ведёт к возможности исполнения моделей и запуска на выполнение действительного кода, сгенерированного по ним. (Код не обязательно должен порождать программное обеспечение, он может

[1]Ранняя версия этой статьи появилась в *Proc. Fundamental Approaches to Software Eng.* (*FASE*), Lecture Notes in Computer Science, vol. 1783, Springer-Verlag, Berlin, Mar. 2000, pp. 22-34.

быть написан на языке описания аппаратуры и приводить к настоящей аппаратуре.) Рис. 1 показывает моделирование системы с полной генерацией кода.

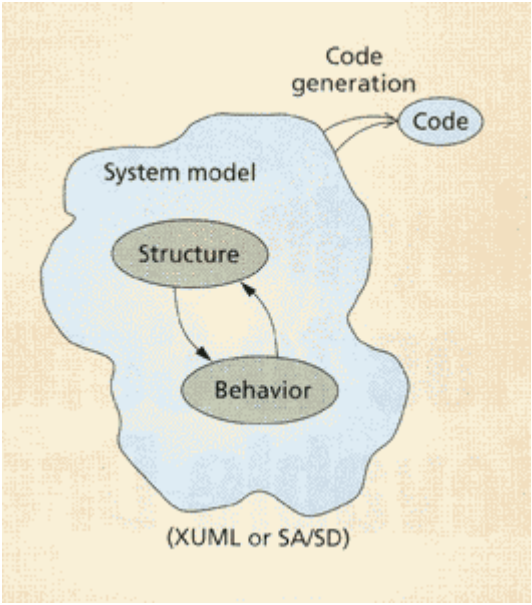

Рис. 1. Моделирование системы с полной генерацией кода (code generation). Модель системы (system model) состоит из структуры (structure) и поведения (behavior), изображённых с использованием визуальных формализмов исполнимого УЯМ (ИУЯМ (XUML)) или структурного анализа / структурного проектирования (СА/СП (SA/SD)). Строгая семантическая основа позволяет автоматически генерировать полностью запускаемый код (code) по модели.

Очевидно, если у нас была бы возможность генерировать полный код, мы бы захотели, в конечном счёте, чтобы этот код послужил основой конечной реализации. Некоторые существующие инструменты, такие как Statemate и Rhapsody от корпорации I-Logix или Rose RealTime от корпорации Rational, могут действительно производить качественный код, достаточно хороший для реализации многих типов реагирующих систем. И нет сомнения в том, что техники этого типа «сверхкомпиляции» высокоуровневых визуальных формализмов со временем будут улучшаться. Обеспечение более высоких уровней абстракции с автоматизированными нисходящими преобразованиями всегда было целесообразным, так как инженеры, делающие фактическую работу, рады работать с абстракциями.

ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ

При разработке сложной системы очень важно иметь возможность протестировать и отладить модель перед тем, как начать интенсивно вкладывать средства в реализацию, – отсюда желание иметь исполнимые модели[1].

Требования являются основой для тестирования и отладки посредством исполнения модели. По природе своей, требования определяют ограничения, желания и надежды, которые мы питаем касательно разрабатываемой системы. Мы хотим убедиться, как во время разработки, так и тогда, когда мы чувствуем, что разработка завершена, в том, что система делает или будет делать то, что мы подразумевали или на что надеялись.

Рис. 2 показывает моделирование системы с полной генерацией кода и гибкими связями с требованиями. Требования могут быть формальными (строго и точно определёнными) или неформальными (письменно, устно, на естественном я зыке или в псевдокоде). Однако, так как эта статья касается, в основном, процессов, которые могут быть автоматизированы, основное внимание уделяется формальным требованиям.

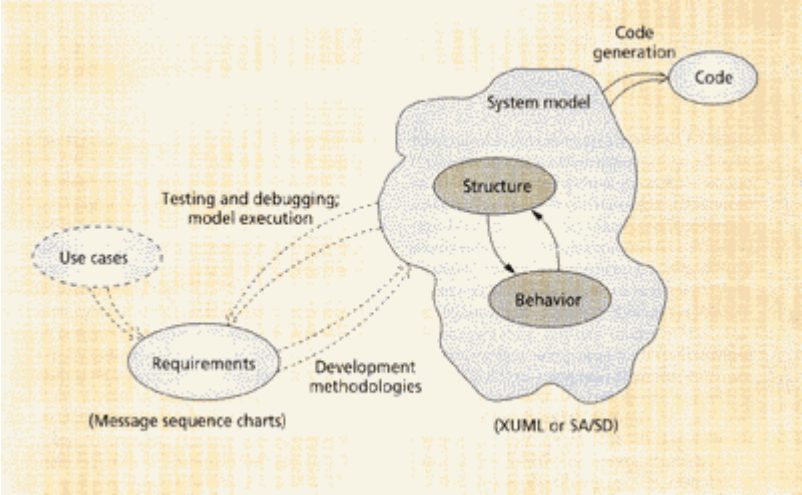

Рис. 2. Моделирование системы с гибкими связями с требованиями (requirements). Разработчики проверяют модель на соответствие требованиям посредством исполнения модели (model execution) (тестирование (testing) и отладка (debugging)). Разработчики используют различные методологии (methodologies) для перехода от требований к модели. В общих чертах, как бы там ни было, эти процессы не всеобъемлющи, не гарантируют достижение успеха и не полностью автоматизированы.

С первых дней появления высокоуровневого программирования компьютерные исследователи ищут ответ на вопрос, как лучше всего сформулировать то, что мы хотим от сложной программы или системы. Значительные усилия включают в себя метод инвариантных суждений Флойда (Floyd) – Хоара (Hoare), с его пред- и постусловиями и операторами завершения[5], и многочисленные варианты временной логики[6].

Эти усилия позволяют выразить два главных типа требований, представляющих интерес при моделировании реагирующих систем. Первое требование – *безопасность*, оно говорит о том, что плохое не может случиться; например, эта программа никогда не завершится с неправильным ответом, или двери этого лифта никогда не откроются между этажами. Второе требование – *живучесть*, оно говорит о том, что должно случаться хорошее. Например, эта программа, в конечном счете, завершится, или этот лифт откроет свои двери на желаемом этаже за выделенное время.

Сценарии и варианты использования

Более новый способ определения требований, который популярен в сфере объектно-ориентированных систем, – использование диаграмм последовательностей сообщений (ДПС). Международный союз по телекоммуникациям (МСТ, бывший МККТТ) принял этот визуальный язык как стандарт очень давно[7]. Идея ДПС проявляет себя и в УЯМ, но в несколько менее явном виде, как язык диаграмм последовательностей[3].

Набор сценариев ДПС не может рассматриваться как реализуемая модель системы. Как будет работать такая система? Что она будет делать при общих динамических обстоятельствах?

И ДПС, и диаграммы последовательностей УЯМ определяют сценарии на последовательности взаимодействий сообщений между объектами. Этот подход очень хорошо сочетается с описанием вариантов использования[8] – неформальной формулировкой возможных способов использования системы. На ранних этапах разработки системы инженеры обычно приходят с описанием вариантов использования, а уже затем определяют сценарии, их иллюстрирующие. Сценарии фиксируют желаемые отношения между процессами, заданиями или объектами, и между этими факторами и внешней средой, линейно или квазилинейно во времени. (Я включаю сюда задания и процессы потому, что, хотя многое в этом обсуждении излагается в терминах объектно-ориентированного подхода и УЯМ, на самом деле в моих аргументах нет ничего особенного для объектов.) Другими словами, разработчик модели использует ДПС для определения сценариев, или «историй», которым конечная система должна и, надо

надеяться, будет удовлетворять, и которые будет поддерживать, и эти сценарии являются иллюстрациями для более абстрактных и общих вариантов использования.

ТРЕБОВАНИЯ ПРОТИВ МОДЕЛИ СИСТЕМЫ

Как показывает рис. 2, описание вариантов использования (use cases) и диаграммы последовательностей не являются частью системы, а, скорее, являются частью требований к системе. Они строятся так, чтобы зафиксировать сценарии, которым мы хотели бы, чтобы удовлетворяла система, когда она будет реализована.

Интересно сравнить межобъектный подход вида «одна история – для всех объектов», который отражают диаграммы последовательностей, с двойным внутриобъектным подходом вида «все истории – для одного объекта», проявляющимся в ИУЯМ-моделировании с использованием диаграмм состояний. В противоположность сценариям, моделирование с диаграммами состояний проводится на более поздних этапах и имеет в результате полное описание поведения для каждого объекта (задания или процесса), обеспечивая детализацию его поведения при всех возможных условиях и для всех возможных историй, предоставленных межобъектными диаграммами последовательностей. Так как это реализуемо напрямую, межобъектное описание находится в центре модели системы на рис. 1 и 2; в конечном счёте, программное обеспечение будет состоять из кода, указанного для каждого объекта.

В противоположность этому, набор сценариев ДПС не может рассматриваться как реализуемая модель системы. Как будет работать такая система? Что она будет делать при общих динамических обстоятельствах? Таким образом, ДПС и диаграммы последовательностей УЯМ обеспечивают требования к поведению, устанавливая, как система должна себя вести, когда она реализована; диаграммы состояний, будучи связанными с диаграммами классов в ИУЯМ, обеспечивают реализуемое поведение как таковое.

В общем и целом, литература не определяет чётко тонкие различия между этими двумя подходами. Снова и снова я нахожу статьи и книги, которые используют одни и те же фразы для представления диаграмм последовательностей и диаграмм состояний. С одной стороны, подобная публикация может гласить, что «диаграммы последовательностей могут использоваться для определения поведения», а ниже – что «диаграммы состояний могут использоваться для определения поведения». Читателю ничего не говорится об основном различии этих двух подходов, – что диаграммы последовательностей являются средством для выражения требований, а диаграммы состояний являются частью модели системы, – или о совершенно различных способах их использования. Наивные читатели часто путаются и попадают в тупик из-за множества типов диаграмм в полном стандарте УЯМ и из-за отсутствия чётких разъяснений касательно того, что же означает описание системы.

«Меж двух огней»

На рис. 2 стрелки между «требованиями» (слева внизу) и «моделью системы» нарисованы пунктиром, так как они не отражают строгие, всеохватывающие, автоматизированные процессы. Переход от требований к модели является долго изучаемой проблемой, и многие методологии разработки систем предоставляют наводящие соображения, эвристику, а иногда и хорошо проработанные пошаговые процессы для этого. Однако какими бы хорошими и полезными эти процессы не были, они являются гибкими методологическими рекомендациями касательно того, как продолжать, а не строгими и автоматизированными методами.

Стрелка, идущая от модели системы к требованиям, изображает собой тестирование и отладку модели по отношению к требованиям и посредством исполнения

модели. Есть хороший способ, как делать это с использованием инструмента Rhapsody. Допустим, что пользователь определил требования в виде множества диаграмм последовательностей, возможно, проиллюстрированных ранее подготовленным описанием вариантов использования. Для простоты будем считать, что результатом является диаграмма *А*. Далее, когда система уже описана в ИУЯМ, пользователь может попросить инструмент Rhapsody запустить модель на исполнение. Во время выполнения, Rhapsody автоматически создаёт анимированные диаграммы последовательностей на лету, показывая динамику взаимодействия объектов, которое проявляется во время выполнения. Допустим, что это отражается на диаграмме *Б*.

Когда выполнение закончено, мы можем попросить Rhapsody сравнить диаграммы *А* и *Б* и выделить любое проявление несовместимости, как-то противоречия в частичном порядке событий, другие различия, вроде событий, имеющихся на одной диаграмме, но отсутствующих на другой. Таким образом, Rhapsody помогает отлаживать поведение системы по отношению к требованиям.

Притом, что это мощный и очень удобный способ проверки поведения модели системы, он ограничен выполняемыми нами запусками и поэтому страдает теми же недостатками, что и классическое тестирование и отладка. Так как система может быть запущена бесконечное число раз, какие-то разы обязательно будут непроверенны, и это могут быть те самые запуски, которые нарушают предъявляемые требования (в этом случае проявится несовместимость с диаграммой *А*). Как постановил Эдгар Дейкстра много лет назад, «тестирование и отладка не могут использоваться для демонстрации отсутствия ошибок, только их присутствия». Эта мягкость процесса отладки является причиной того, что стрелка от модели системы к требованиям на рис. 2 также проведена пунктиром.

ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТЕЙ И ДИАГРАММЫ ЖИВЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Как языки описания требований, все известные версии ДПС, включая стандарт МСТ и диаграммы последовательностей, принятые в УЯМ, являются крайне слабыми по выразительности. Их семантика лишь немного больше, чем множество простых ограничений на частичный порядок возможных событий при исполнении системы. Фактически, в ДПС ничего нельзя сказать о том, что же будет делать система на самом деле, когда она запущена. Эти диаграммы могут сформулировать то, что может случиться, но не то, что должно случиться. Таким образом, удивительно, если быть точным, по большинству определений семантики ДПС пустая система – та, которая ничего не делает в ответ на всё, что угодно, – удовлетворяет любой такой диаграмме. То есть сидение и ничегонеделание будет удовлетворять вашим требованиям. (Обычно, однако, есть минимальное, часто не выраженное явно требование того, чтобы хотя бы один запуск системы прошёл по одной указанных диаграмм последовательностей.)

Другой трудностью ДПС является их неспособность определения нежеланных сценариев. Мы хотим запретить проявление этих антисценариев, а они являются решающими при установке требований по безопасности.

Одна из последних работ была посвящена этим недостаткам, и в ней было предложено расширение ДПС, называемое диаграммами живых последовательностей (ДЖП)[9]. Как говорит само название, ДЖП определяют живучесть, то, что должно произойти. Они позволяют разработчику модели различать возможное и необходимое поведение, как глобально, на уровне всей диаграммы, так и локально, при указании событий, условий и прогресса во времени внутри диаграммы. Живые, или горячие, элементы также позволяют определять антисценарии. Другие элементы, называемые холодными, поддерживают ветвления и итерации.

ДЖП позволяют начать смотреть более серьёзно на двойственость реагирующего поведения – связь между взглядом на межобъектные требования и внутриобъектную модель системы.

Демонстрация того, как ДЖП обращаются с условиями или ограничениями, даёт представление о том, как они работают. Допустим, что *П* – это горячее условие, появляющееся в определённом месте на диаграмме. Тогда *П* должно быть истинно всегда, когда это место достигается в процессе работы системы, а если оно ложно, то система прекращает свою работу. Другими словами, *П* на самом деле должно быть истинно; иначе имеет место непростительная ошибка. Таким образом, разработчики модели могут определить антисценарии (двери лифта открываются тогда, когда они не должны, или ракета запускается до того, как радар захватит цель). В противоположность этому, если *П* является холодным условием, тогда оно тоже должно быть истинно всегда, когда указанное на диаграмме место достигается, но если оно ложно, то это не катастрофа. Скорее, происходит выход, и мы передвигаемся на один уровень вверх – из текущей диаграммы, если *П* находится на его верхнем уровне, или из поддиаграммы, продолжая на внешнем уровне, если *П* находится внутри блока поддиаграммы. Это позволяет определять контролирующие конструкции, такие как если-то-иначе и пока-делать с использованием *П* в качестве контролирующего ограничения.

Ещё не ясно, являются ли ДЖП тем самым, что нам точно нужно, и определённо потребуется дополнительная работа. Нам надо набраться опыта с использованием этого языка, и мы должны делать хорошие реализации. Тем не менее, относительно ДПС, ДЖП предлагают гораздо более мощный способ визуального определения поведенческих требований. Так как они обладают значительно большей выразительностью (по существу, это касается самого ИУЯМ), ДЖП также позволяют начать смотреть более серьёзно на двойственность реагирующего поведения – связь между взглядом на межобъектные требования и внутриобъектную модель системы.

ПРОВЕРКА И СИНТЕЗ

На рис. 3 две пунктирные стрелки между требованиями и моделью стали сплошными. Это означает, что в нашем распоряжении есть жёсткие, формальные и строгие – и почти полностью автоматизированные – связи между моделью системы (на ИУЯМ[4], например, или на подходящей версии СА/СП) и требованиями (в виде ДЖП[9], например, или во временной логике, или в виде временных диаграмм[10]).

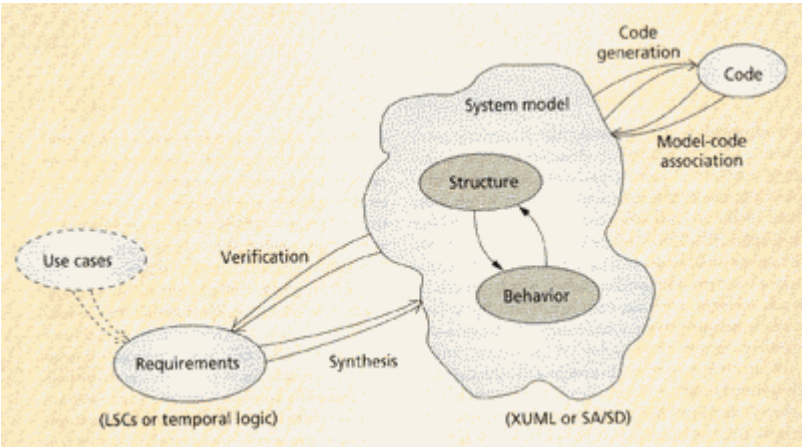

Рис. 3. Моделирование системы с жёсткими связями с требованиями. Это – главная часть мечты, в которой есть истинная проверка (verification) на соответствие модели требованиям, и синтез (synthesis) идёт от требований к модели. Эти процессы, являясь доступными, будут всеобъемлющими, гарантирующими достижение успеха и полностью автоматизированными.

От модели к требованиям

При переходе от модели системы к требованиям, вместо тестирования и отладки посредством запуска моделей, мы заинтересованы в использовании истинной проверки модели системы на соответствие требованиям. Это не тот CASE-инструмент, который люди в 1980-х годах часто называли «аттестацией и проверкой», – он не представлял собой чего-то много большего, чем просто проверка согласованности синтаксиса модели. Скорее, это математически строгое и точное доказательство того, что модель удовлетворяет требованиям, и это доказательство проводит автоматизированное средство проверки.

Так как мы используем мощные языки вроде ДЖП (или аналогичную временную логику, или временные диаграммы), это требует больше, чем просто исполнения модели системы и убеждённости в том, что полученные в результате запуска диаграммы последовательностей являются согласованными с теми, что вы подготовили заранее. Это означает убеждённость в том, например, что, события, отмечаемые ДЖП как те, которые не должны произойти (антисценарии), действительно никогда не случатся, а те, которые должны произойти (или должны произойти в течение определённых промежутков времени), действительно будут иметь место. Это те самые факты, которые, в общем случае, никакой объём запусков на выполнение не может проверить.

Хотя полная проверка представляет собой разрешимую алгоритмическую проблему, сама идея строгой проверки программ и систем – аппаратного и программного обеспечения – прошла долгий путь от первых работ по инвариантным суждениям и более поздних работ по временной логике и проверке моделей. В наши дни мы можем с уверенностью сказать, что мы можем провести истинную проверку во многих случаях, особенно в случаях с конечным числом состояний, имеющих место в реагирующих системах реального времени.

Компания I-Logix недавно произвела на свет версию инструмента Stalemate с возможностями проверки, который будет выпущен как продукт в ближайшем будущем. Осуществление того же самого для ООАП-инструмента вроде Rhapsody – это всего лишь вопрос времени. Вскоре, мне верится, мы будем в обычном порядке использовать автоматизированные инструменты для проверки моделей на соответствие требованиям.

От требований к модели

В обратном направлении, при переходе от требований к модели, мы имеем синтез. Вместо того, чтобы вести разработчиков системы неформальными путями для получения моделей в соответствии с их желаниями и надеждами, мы хотели бы, чтобы наши инструменты могли синтезировать прямо из этих самых желаний и надежд, если они, на самом деле, реализуемы. Мы хотим иметь возможность автоматически генерировать модель системы по требованиям. (В контексте этого обсуждения я предполагаю, что структура – разбиение на объекты или компоненты, например, – уже была определена.)

Это гораздо труднее, чем синтез кода по модели системы, которое, на самом деле, является всего лишь высокоуровневым видом компиляции. Двойственность между стилем сценария (требования) и стилем диаграмм состояний (моделирование) в контексте сказанного о том, что система делает на протяжении времени, превращает синтез реализуемой модели системы по представленным в виде последовательностей требованиям в по-настоящему сложную задачу. Это не очень трудно сделать для слабых ДПС, которые не могут многого сказать о том, что мы в действительности хотим от системы. Это гораздо сложнее для более реалистичных языков требований, вроде ДЖП или временной логики.

Как тогда мы может синтезировать хорошее первое приближение диаграмм состояний по ДЖП? Несколько исследователей обращали своё внимание на схожие

проблемы, результатом чего стали работы по отдельным типам синтеза по временным диаграммам[10] и временной логике[11]. Одна из последних работ описывает первую попытку создания алгоритмов синтеза конечных автоматов и диаграмм состояний по ДЖП (хотя в несколько ограниченной постановке и пока неприменимую для очень больших моделей)[12]. Метод сначала определяет, являются ли требования действующими (удовлетворяет ли им какая-либо существующая модель системы), а затем использует тот факт, что бытие действующими и обладание моделью (бытие реализуемыми) для требований являются равнозначными основаниями для синтеза действительной модели.

Значительно более глубокие исследования по этому поводу в настоящее время ещё проводятся. Я верю, что синтез, в конечном счете, закончится так же, как и проверка, – основательно в принципе, но не далеко от практического и полезного решения.

От кода к модели

Рис. 3 также содержит сплошную стрелку, идущую от кода к модели системы («связь модель – код» (model-code association) вверху справа). Это означает способность разработчика перейти назад к от кода к модели. Осуществление определённых типов изменений в коде автоматически отражается назад как изменения в визуальных формализмах модели. Это воспроизводит классический цикл действий, имеющих место между проектированием и реализацией, легче и менее ошибочно. Rhapsody предоставляет полезную форму этой связи модель – код. Есть причины верить, что эта способность будет обычным делом в будущем, и что применимость техник, делающих это возможным, станет шире и мощнее.

КАК СЛЕДУЕТ ПРОДОЛЖАТЬ РАЗРАБОТКУ?

Рис. 3 подразумевает, что нам вообще не понадобятся стрелки, идущие справа налево, и разработчики могут работать без проверки, тестирования или без связи модель – код. Разработчик системы может перейти прямо и ровно от желаний к результатам: сформулируйте Ваши требования, используйте свой инструмент для синтеза модели системы, используйте его же для генерации кода по модели, и всё.

Очевидно, это не тот случай. Мы всегда будет нуждаться в пошаговой разработке систем, с различными имеющимися циклами действий, возможно, в соответствии со спиральной философией разработки. Такая методология предусматривает циклы разработки, производящие непрерывно совершенствуемые и улучшаемые версии системы. Один из циклов мог бы быть между требованиями и моделью, пошагово улучшающий и совершенствующий разрабатываемую систему путём следования пунктирным стрелкам на рис. 2 – методологиям разработки, тестирования и отладки, и сплошным стрелкам на рис. 3 – синтезу и проверке. Другой (менее значимый) цикл мог бы быть таким же, как и цикл требования – модель, но он бы был между моделью и реализацией в коде, раз за разом улучшающий конечный артефакт.

Так как нам, в конечном счёте, может понадобиться внести изменения в классический жизненный цикл для достижения чего-то, я не выработал полной пошаговой методологии того, как продолжать разработку системы. Вместо этого, эта статья описывает различные варианты такой методологии, языки и инструменты, которые они используют, а также их взаимоотношения. Чтобы предложить полновесную методологию, нам потребуется больше, чем просто несколько примеров в наспех написанных книгах по методологии, которые полны мудрости, но технически пусты. Это не свершится сразу и вдруг. Скорее, нам надо будет положиться на глубокие знания и навыки, которые будут накапливаться за годы опыта по использованию этих техник с их полным смысловым обоснованием и поддержкой со стороны поистине мощных инструментов.

ИГРОВЫЕ СЦЕНАРИИ

Чтобы ту мечту, которую я обрисовал, сделать полной, нам понадобится один последний кусочек: значительно более удобный способ установки поведенческих требований, приемлемых не только для системных инженеров, но также и для их клиентов, как-то пользователей и контрагентов.

С этой стороны я предлагаю игровые сценарии. Когда Вы исполняете модель, Вы проигрываете некий сценарий. Это становится очевидно, когда Вы используете инструмент для выполнения моделей в диалоговом режиме. Это становится особенно ясным и впечатляющим (а заодно и полезным), когда Вы работаете с программной моделью конечного интерфейса системы или даже с фактической аппаратурой системы, что возможно для ранее упомянутых инструментов. Вы можете отыграть некий сценарий как непосредственный участник, так сказать, для окружения системы, внося события и изменения в значениях и наблюдая результаты по мере их появления[1].

Вместо того, чтобы использовать традиционные языки для определения сценариев, модельщики работают напрямую с моделью интерфейса системы, используя высокодружественный пользователю метод обучения их инструмента желанным и нежеланным сценариям.

В противоположность этому здесь мы будем играть в сценарии. Это делается до построения какой-либо поведенческой модели системы, чтобы установить требования, возможно, управляемые вариантами использования. Вместо того, чтобы использовать традиционные языки, будь то визуальные или другие, для определения сценариев, модельщики работают напрямую с моделью интерфейса системы, используя высокодружественный пользователю метод обучения их инструмента желанным и нежеланным сценариям. Разработчики могут делать эту работу вместе с клиентами или потенциальными пользователями, облегчая сам процесс и устраняя многие типы поведенческих ошибок, которые всплывают в процессе разработки.

Представьте себе графическое изображение сотового телефона, например, появляющееся на экране компьютера разработчика. За ним ничего нет. Для него не было ещё определено никакого поведения. Теперь Вы начинаете вводить сценарии посредством щелчков и перетаскивания мышью, играя с входами и реакциями системы, указывая, являются ли те или иные элементы горячими или холодными, особенными для данного экземпляра или общими, и т.д. Повторяющийся процесс также включает средства улучшения структуры системы во времени путём формирования составных объектов и их совокупностей и установки наследующих объектов.

По мере продолжения игры в основанные на сценарии требования низлежащий инструмент – игровой движок – будет автоматически и пошагово генерировать формальные ДЖП (не только ДПС) или формулы временной логики, которые точно фиксируют игровые сценарии. Вместо использования абстрактных инженерно-ориентированных языков, мы применяем дружественный, интуитивный, пользовательско-ориентированный автоматизированный процесс для конструирования строгих и исчерпывающих требований.

Здесь также предстоит провести множество исследований. Так как идея игровых сценариев имеет непростую математико-алгоритмическую сторону, большая часть необходимых усилий связана с её человеческими аспектами. Должен существовать мощный, при этом естественный и легкоиспользуемый способ взаимодействия с по существу свободной от поведения «системной оболочкой», чтобы мы могли сказать её то, что мы хотим от неё. Я и один из докторантов разрабатываем первую версию игровой среды в Институте Вейцманна и надеемся вскоре опубликовать подробности.

Рис. 4 подытоживает полную мечту разработки систем.

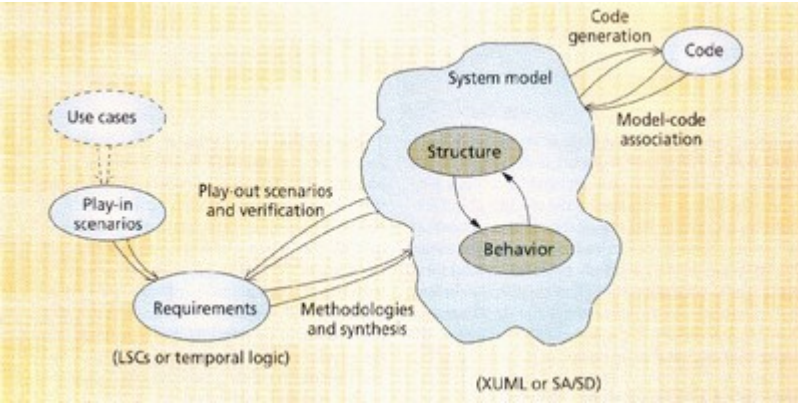

Рис. 4. Мечта полностью. В дополнение к проверке и синтезу мечта предусматривает «разыгрывание» требований напрямую с моделью интерфейса системы, используя дружественный пользователю метод обучения системы её поведению. Разработчики могут разыгрывать сценарии вместе с клиентами или потенциальными пользователями.

Возможно, не будет большим преувеличением сказать, что есть гораздо больше того, чего мы не знаем и не можем пока достичь в этом деле, чем того, что мы знаем и можем достичь. В дополнение к представленным мною темам, много проблем требуют дальнейших исследований и разработок, чтобы удовлетворительно вписаться в общую схему. Среди них – анализ в реальном времени, автоматическое расположение диаграмм и взаимодействие с гибридными системами, имеющими как непрерывные, так и дискретные аспекты.

Усилия множества исследователей, методистов и проектировщиков языков привели к большим результатам, чем мы могли надеяться десятилетие или примерно столько назад, и за это мы должны быть благодарны и смиренны. Впереди ещё длинная дорога, но на горизонте есть мечта. Хотя несколько частей этой мечты даже не близки к полной доступности, сама мечта не является недосягаемой. Если эта мечта осуществится, она даст значительный эффект при разработке сложных систем.☼

Ссылки:

1. D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development," *Computer*, Jan. 1992, pp. 8-20.
2. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, 1987, pp. 231-274; also tech. report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, 1984.
3. Unified Modeling Language (UML) documentation, Object Management Group (OMG), http://www.omg.org/.
4. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, July 1997, pp. 31-42.
5. A. Apt, *Verification of Sequential and Concurrent Programs*, 2nd ed.. Springer-Verlag, New York, 1997.
6. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Veriag, New York, 1992.
7. "MSCs: ITU-T Recommendation Z.120: Message Sequence Chart (MSC)," ITU-T, Geneva, 1996.
8. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley. Reading, Mass., 1992.
9. W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, submitted for publication. An early version was published in *Proc. 3rd IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems*, P. Ciancarini, A. Fantechi, and R. Gorrieri, eds., Kluwer Academic, New York, 1999, pp. 293-312.
10. R. Senior and W. Damm, "Specification and Verification of System-Level Hardware Designs Using Timing Diagrams," *Proc. European Conf. Design Automation*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 518-524.
11. A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proc. 16th ACM Symp. Principles of Programming Languages*, ACM Press, New York, 1989, pp. 179-190.
12. D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," *Proc. 5th Int'l Conf. Implementation and Application of Automata*, Lecture Notes in Computer Science, Springer-Verlag, New York, submitted for publication; also, tech. report MCS99-20, The Weizmann Institute of Science, Rehovot, Israel, Oct. 1999.

***Давид Харел (David Harel)** является профессором Уильяма Суссмана (William Sussman) в Научном институте Вейцманна, что в Израиле, и деканом факультета математики и информатики. Он также является сооснователем и главным учёным в компаниях I-*

Logix и DigiScents Israel. К его научно-исследовательским интересам относятся теория исчислимости и сложности, программные логики, теория автоматов, визуальные языки, системотехника и, самое недавнее, математика и алгоритмы обонятельного общения. Харел (Harel) получил степень доктора философии (PhD) по информатике в Массачусеттском технологическом институте (Massachusetts Institute of Technology). Его последними книгами стали: «Ограниченные компьютеры: что они действительно не могут делать» (Computers Ltd.: What They Really Can't Do) *(издательство Oxford University Press, Лондон, 2000 г.) и* «Динамическая логика» (Dynamic Logic) *(в соавторстве с Декстером Козеном (Dexter Kozen) и Джуреком Тиурином (Jurek Tiuryn), издательство MIT Press, Кембридж, Mass., 2000 г.). Харел (Harel) является членом Ассоциации по вычислительной технике (Association for Computing Machinery, ACM) и Института инженеров по радиотехнике и электронике (Institute of Electrical and Electrolnic Engineers, IEEE). С ним можно связаться по адресу: harel@wisdom.weizmann.ac.il.*

Структурный анализ и структурное проектирование

Структурный анализ и структурное проектирование, СА/СП (Structured Analysis and Structured Design, SA/SD), появившиеся в конце 1970-х гг., основываются на подъёме классических процедурных программных концепций на уровень моделирования и использовании диаграмм, или визуальных формализмов, в качестве языков для моделирования структур систем. Структурные модели основываются на функциональной декомпозиции и потоках информации и рисуются в виде иерархических диаграмм потоков данных.

Многие методисты способствовали основанию фундамента для теории СА/СП, придумав структуру функциональной декомпозиции и диаграмм потоков данных, среди них – Том Демарко (DeMarco)[1], Ларри Константайн (Larry Constantine) и Эд Йордон (Ed Yourdon)[2]. Длительная работа Дэвида Парнаса (David Parnas) также имела большое значение.

В середине 1980-х гг. три методические команды – Уарда и Меллора (Ward and Mellor)[3], Хэтли и Пирбая (Hatley and Pirbhai)[4] и команда «Мёртвая точка» (Stalemate)[5] – расширили эту базовую модель СА/СП посредством использования диаграмм состояний[6] и их более богатого языка для добавления поведения к этим достижениям. Диаграмма состояний связана с каждой функцией или действием и описывает её поведение. Много непростых проблем предстояло разрешить, чтобы должным образом связать структуру с поведением, позволив модельщикам конструировать полную и семантически правильную модель системы. Недостаточно просто выбрать поведенческий язык, а затем связать каждую функцию или действие с поведенческим описанием. (Это было бы похоже на то, как сказать, что всё, что Вам нужно при конструировании автомобиля, – это структурные составляющие: корпус, шасси, колёса и т.д., плюс двигатель, после чего Вы просто помещаете двигатель под капот – и всё.) Упомянутые три команды поролись с этой проблемой, и их решения по поводу того, как связать структуру с поведением, оказались очень похожи. Тщательное поведенческое моделирование и его тесная связь со структурой системы являются особенно важными для реагирующих систем[7,8], среди которых системы реального времени представляют особый случай.

Продукт Stalemate, выпущенный в 1987 г., стал первым коммерческим инструментом с возможностью исполнения моделей и кодогенерации по высокоуровневым моделям[5] (http://www.ilogix.com/). Книга «Моделирование реагирующих систем с использованием диаграмм состояний: подход Statemate» *(Modeling Reactive Systems with Statecharts: The STATEMATE Approach)*[9] даёт обновлённое и подробное изложение языков СА/СП, их взаимоотношений и способа их реализации в продукте Statemate.

Конечно, модельщики не обязаны принимать конечные автоматы или диаграммы состояний для описания поведения. Существует много других языков, которые можно связать со структурными диаграммами СА/СП. Среди них такие визуальные формализмы, как сети Петри или диаграммы упрощённых связей данных (Simplified Data Link, SDL), а также больше алгебраические методы вроде сообщения последовательных процессов (Communicating Sequential Processes) или исчисления сообщающихся систем (Calculus of Communicating Systems).

Ссылки

1. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.
2. L.L. Constantine and E. Yourdon, *Structured Design*, Prentice Hall, Upper Saddle River, N.J., 1979.
3. P. Ward and S. Mellor, *Structured Development for Real-Time Systems: Volumes 1-3*, Yourdon Press, New York, 1985.
4. D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1987.
5. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Soft. Eng.*; vol. 16, no. 4,1990, pp. 403-414; also in *Proc. 10th Int'l Conf. Soft. Eng.*, IEEE Press, Piscataway, N.J., 1988, pp. 396-406.

6. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8,1987, pp. 231-274; also tech. report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, 1984.
7. D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logics and Models of Concurrent Systems*, K.R. Apt, ed., NATO AS1 Series, vol. F-13, Springer-Verlag, New York, 1985, pp. 477-498.
8. A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," *Current Trends in Concurrency*, J. de Bakker et al., eds.. Lecture Notes in Computer Science, vol. 224, Springer-Verlag, Berlin, 1986, pp. 510-584.
9. D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, New York, 1998.

Объектно-ориентированный анализ и проектирование

Конец 1980-х гг. увидел первые предложения в области объектно-ориентированного анализа и проектирования, ООАП (Object-Oriented Analysis and Design, OOAD). Как и в СА/СП, главной идеей при моделировании структур систем был подъём концепций с программного уровня на уровень моделирования и использование визуальных формализмов. Будучи вдохновлёнными диаграммами «сущность – связь»[1], несколько методических команд рекомендовали различные формы диаграмм классов и объектов для моделирования структур систем[2-5]. Для моделирования поведения большинство объектно-ориентированных подходов к моделированию приняли диаграммы состояний[6]. Каждый класс имеет связанную с ним диаграмму состояний, которая описывает поведение любого объекта – экземпляра класса.

Проблема соединения структуры и поведения для ООАП более тонка и сложна, чем для СА/СП. Классы представляют собой динамически меняющиеся коллекции конкретных объектов. Поэтому поведенческое моделирование должно обращаться и с такими проблемами, как создание и уничтожение объектов, передача сообщений, изменение и поддержание связей, сборка, наследование и т.д.

Связи между поведением и структурой должны быть определены на достаточном уровне детализации и достаточно строго, чтобы поддерживать создание инструментов, позволяющих осуществлять исполнение моделей и полную кодогенерацию. Всего лишь несколько инструментов умеют делать это. Один из них – Object-Time, основанный на методе объектно-ориентированного моделирования в реальном времени (Real-Time Object-Oriented Modeling)[5] и в настоящее время являющийся частью инструмента Rational RealTime (см. http://www.rational.com/).

Другим инструментом является Rhapsody (http://www.ilogix.com/), который основан на работе Эрана Джери (Eran Gery) и Давида Харела (David Harel) по исполнимому объектному моделированию с использованием диаграмм состояний[7]. Эта работа фокусируется на тщательно созданном языковом наборе, который включает диаграммы классов/объектов, адаптированные из метода Буча (Booch) и метода объектного моделирования (Object Modeling Technique, OMT), управляемые диаграммами состояний для описания поведения.

Эта пара языков также служит исполнимым ядром унифицированного языка моделирования (Unified Modeling Language, UML)[8], собранного воедино командой, возглавляемой Гради Бучем (Grady Booch), Джеймсом Румбау (James Rumbaugh) и Айваром Якобсоном (Ivar Jacobson), и принятого группой Object Management Group в качестве стандарта в 1997 году (см. http://www.omg.org/). Диаграммы классов/объектов и диаграммы состояний как часть UML часто называется XUML (eXecutable UML – исполнимый UML). Таким образом, XUML является частью UML, которая определяет однозначные, исполнимые и поэтому реализуемые модели.

UML имеет несколько средств для определения более сложных аспектов структур и архитектур систем (например, блоки и компоненты). Важной частью UML для определения требований являются варианты использования Якобсона[9].

Ссылки

1. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Trans. Database Systems*, vol. 1, no. 1, 1976, pp. 9-36.
2. G. Booch, *Object-Oriented Analysis and Design, with Applications*, 2nd ed., Benjamin-Cummings, San Mateo, Calif., 1994.
3. S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Prentice Hall, Upper Saddle River, N.J., 1994.
4. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, Upper Saddle River, N.J., 1991.
5. B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.
6. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8,1987, pp. 231-274; also tech. report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, 1984.

7. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, July 1997, pp. 31-42.
8. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Reading, Mass., 1999.
9. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, Reading, Mass., 1992.