Применение формальных методов при спецификации бизнес-процессов

Морковкин Василий

2022

Аннотация

История отрасли информационных технологий насчитывает немало случаев масштабных нарушений работы из-за изъянов программного обеспечения. Изъяны могут нести финансовые и репутационные потери, а также ставить под угрозу безопасность персональных данных и жизнь человека. Поэтому неотъемлемой частью процесса разработки программного обсепечения является поиск ошибок и их исправление.

Ошибки могут появляться на этапах:

- спецификации,
- написания кода.

Практика покрытия кода автоматизированными тестами помогает минимизировать ошибки этапа написания кода, хорошо описана и широко применяется в индустрии. Ошибки спецификации, однако, такими тестами не обнаруживаются. Они могут проявляться в виде нарушения инваринтов работы системы и даже противоречивости постановки задачи. О возможных противоречиях в требованиях и теоретически достижимых гарантиях любой системы лучше знать еще до начала ее разработки.

С этой целью данная работа фокусируется на разработке, эксплуатации и анализе применимости метода спецификации бизнес-процессов. За основу берутся формальные методы к верификации. Метод должен осваиваться разработчиками за разумное время, а результат его применения оправдывать расходы на использование.

Оглавление

Введение

В данной главе будут введены основные понятия, сформулированы цели, приведен анализ существующих подходов и описана структура работы.

1.1 Основные понятия

В контексте данной работы введем следующие понятия.

Формальные методы — набор математических техник, применяемых в сфере информационных технологий для формализации рассуждений и построения систем с изученными свойствами.

Спецификация — процесс разработки технического задания, которое может быть транслировано разработчиками в программный код.

Бизнес-процесс — упорядоченный набор действий, выполняемых людьми или машинами, результатом исполнения которых является продукт или услуга, потребляемые пользователями.

1.2 Цели работы

- Исследование существующих подходов к валидации спецификаций бизнес-процессов.
- Разработка методики спецификации бизнес-процессов с применением формальных методов.

- Подготовка инструментария для использования методики в продуктовой разработке.
- Применение методики на практике, оценка затрат, преимуществ и ограничений.

1.3 Актуальность работы

Ошибки программного обеспечения могут угрожать жизни людей и оборачиваться значительными финансовыми убытками. К примеру, редкая ошибка конкуррентности приводила к неконтролируемому разгону машин Toyota [1]. А из-за гонки на данных аппарат лучевой терапии Therac-25 генерировал небезопасные дозы излучения [2].

Примеры показывают, что даже инвестиция значительного времени в тестирование продукта не позволяет убедиться в отсутствии ошибок конкурентности. Проблема усугубляется тем, что писать надежные конкурентные тесты - весьма нетривиальная задача [3]. Более того, сам код может соответствовать спецификации, в то время как спецификация может допускать исполнения, ведущие к нежелательным последствиям.

Дело в том, что люди разрабатывают сложные системы, где число состояний огромно. Занимаясь ручным перебором возможных исполнений легко что-то упустить. Однако, призвав на помощь опыт математиков, о системах можно рассуждать в терминах их свойств. Такие свойства можно формулировать в утверждениях вида:

- “В системе никогда не происходит событие X“,
- “Если произошло X, то в конце концов произойдет и Y“,
- “Работа алгоритма завершается“, и так далее.

Такие свойства звучат естественно и понятны людям без специальных знаний. Имея механизм проверки этих свойств, можно строить системы, дизайн которых обладает предсказуемым поведением. Подобный подход получил рас-

пространение в сфере компьютерной безопасности [4] и алгоритмов распределенных систем [5].

При разработке бизнес-процессов спецификации зачастую выглядят как текст на натуральном языке (русский, английский и т.д.). В лучшем случае, с применением нотаций вроде BPMN [6] и UML [7]. Проблема таких спецификаций в том, что они не делают акцент на описании и поддержки инвариантов процессов, которые описывают. На момент написания статьи автору не удалось найти фреймворка, который бы помогал в проверке дизайна на непротиворечивость и отсутствие изъянов, и в то же время был бы доступным для освоения разработчиками или системными аналитиками. Наверняка, владелец любого продукта не отказался бы от понимания того, что в его системе произойти не может, и что непременно произойдет. Так почему бы не попробовать формальные методы в продуктовой разработке?

1.4 Анализ подходов

Идея применения формальных методов для спецификации бизнес-процессов не нова. Существующие фреймворки берут на вооружение различные формализмы: *императивный*, *декларативный*, *событийный* и *артефактный*.

В *императивном* формализме процессы моделируются как множества *задач* или *активностей*, *вентилей (анг. gates)*, и *событий*, связанных *потоками (анг. flows)* или *переходами (анг. transitions)*. Каждая активность описывает единицу работы, а переходы описывают порядок между единицами работы. Императивные подходы включают в себя верификацию распространенных нотаций BPMN [8], BPEL [9], UML [10] и YAWL [11].

С другой стороны, *декларативыный* формализм не прибегает к концепции потоков, которые определяют очередность операций. Вместо этого процесс моделируется как множество *активностей* и множество *ограничений* на очередность этих активностей. Любое исполнение процесса, не запрещенное этими

ограничениями, считается корректным.

Событийный формализм [12] - еще один подход к моделированию. Его основу составляют *событийные процессные цепочки* — направленные графы, состоящие из *событий*, *активностей* и *вентилей*. В отличие от императивного подхода, в событийном подходе нет явного моделирования порядка операций.

Наконец, *артефактный* формализм фокусируется на эволюции бизнес-сущностей и данных. Такие спецификации включают в себя понятие жизненного цикла бизнес-сущностей (артефактов), таких как данные.

Для целей работы больше подходит декларативный подход, поскольку он не привязан к конкретной нотации. Более того, его применение не требует уже готовой спецификации. Отталкиваясь от набора простых желаемых свойств, он может выступать в роли интерактивного ассистента при разработке спецификации.

1.5 Структура работы

В Главе 2 рассматриваются математеческие формализмы, пригодные для формирования декларативной методики. Поясняется выбор языка спецификаций TLA$^+$. Описываются его базовые конструкции и выразительные возможности.

В Главе 3 разрабатывается сама методика. Приводятся детальные примеры использования и анализ опыта реального использования в продуктовой разработке.

В Глава 4 описывается разработка инструментария TLA$^+$, полезного для применения методики.

Работа завершается подведением итогов и обсуждением дальнейших путей развития.

Выбор инструмента

В данной главе будет произведен обзор формальных методов *доказательства теорем* и *проверки моделей*, а также пояснен выбор в пользу проверки моделей для целей данной работы. Затем будет обозначена необходимость и способы моделирования времени, проведено сравнение наиболее популярных инструментов проверки моделей *Alloy* и TLA$^+$. И, наконец, описаны концепты TLA$^+$, которые будут использоваться в дальнейшем.

2.1 Сравнение технологий

В плеаде формальных методов ярко выделяются два похода: *доказательство теорем (анг. theorem proving)* и *проверка моделей (анг. model checking)*.

В доказательстве теорем верификация устроена индуктивно, шаг за шагом: отталкиваясь от выбранной системы аксиом и правил вывода выводятся комплексные утверждения. При использовании доказательного ассистента нам доступна работа с более точными представлениями наших систем. Однако, занимаясь проблемой разрешимости (нем. Entscheidungsproblem), Алан Тьюринг показал [13], что не может существовать завершающегося алгоритма, который бы в качестве входных данных принимал утверждение некоторого формального языка, а на выходе выдавал бы один из двух ответов: “истина“ или “ложь“. Следствием этого фундаментального результата является необходимость проводить доказательства вручную (за исключением ограниченного набора простых

случаев), а это требует значительной экспертизы.

В подходе проверки моделей мы описываем абстрактную версию системы и можем автоматически (с помощью *проверщика моделей (анг. model checker)*) проверить ее на соответствие свойствам. При этом наша модель должна быть достаточно маленькой (в терминах количества состояний), чтобы быть обработанной проверщиком за разумное время.

Поскольку нашей целью является создание методики, доступной для освоения широким кругом людей без должной математической подготовки, фундаментом методики послужит *проверка моделей*.

2.2 Моделирование времени

Нам интересна проверка утверждений вида “X всегда верно“, “В конце концов X будет верно“, “X верно до тех пор, пока не верно Y“, “Если X, то в конце концов Y будет врно“. Эти утверждения строятся вокруг концепта времени. Значит, нужен способ для его моделирования. Подходящим формализмом является *темпоральная логика*. Ее разновидность, *линейная темпоральная логика*, как раз позволяет кодировать утверждения о будущих путях исполнения. Типичными операторами являются:

- всегда $(\Box F)$
- в итоге $(\Diamond F)$
- если было P, то будет и Q $(P \rightsquigarrow Q)$

Наиболее развитыми языками спецификаций с возможностями проверки моделей и рассуждения о времени с помощью линейной темпоральной логики являются TLA$^+$ [14] и Alloy версии 6 [15].

На момент написания в Alloy поддержка темпоральной логики появилась недавно, не покрыта документацией и примерами использования. Поэтому, хоть Alloy и является перспективным инструментом верификации с акцентом на визуализацию, в данной работе будет использоваться TLA$^+$.

2.3 Возможности TLA$^+$

Большинство математических концептов, используемых в TLA$^+$, просты. Данная секция дает представление о том, как выглядит написание выражений в TLA$^+$.

Зачастую языки программирования имеют свою нотацию для математических оперторов. Например, логические операторы Java ($!, \&\&, ||$) в математике обозначаются как ($\neg, \vee, \wedge$) и отсутствует на большинстве раскладок клавиатур. По этой причине логический операторы в TLA$^+$ записываются как (~, /\, \/). Для других математических символов используются их эквиваленты из LaTeX.

2.3.1 Логика высказываний

TLA$^+$ строится поверх *логики высказываний*. Высказывание — это утверждение на булевых переменных, которые принимают значения True или False. К примеру, выражение

$$A \wedge (B \vee \neg C)$$

означает “A верно И (B верно ИЛИ C ложно)“, а выражение

$$A \Rightarrow B = \neg A \vee B$$

называется “импликацией“ и эквивалентно утверждению “либо A ложно, либо B истинно“. Равенство в логике высказываний определяется как импликация в обе стороны.

TLA$^+$ умеет вычислять высказывания:

Q : FALSE $\Rightarrow$ FALSE

A : TRUE

Q : FALSE $\Rightarrow$ FALSE

A : TRUE

Например, используя генераторы множеств, можно вывести таблицу истинности для импликации:

$Q : \{\langle A,\ B,\ A \Rightarrow B\rangle : A,\ B \in \text{BOOLEAN}\ \}$

$A : \{\langle$FALSE, FALSE, TRUE$\rangle$,

$\langle$FALSE, TRUE, TRUE$\rangle$,

$\langle$TRUE, FALSE, FALSE$\rangle$,

$\langle$TRUE, TRUE, TRUE$\rangle\}$

2.3.2 Теория множеств

Следующим важным формализмом TLA$^+$ является *теория множеств*. Несколько примеров:

- $\{1, 2\}$ - фигурные скобки в качестве конструктора,
- $1..N$ - множество от 1 до N,
- $x \in S$ - проверка принадлежности x множеству S,
- $S \times S$ - декартово произведение множества S с самим собой.

Множества можно фильтровать. Выражение $\{x \in S : P(x)\}$ является множеством всех элементов S, для которых истин предикат $P(x)$:

$Q : \{\langle x,\ y\rangle \in \{1,\ 2\} \times \{3,\ 4\} : x > y\}$

$A : \{\langle 1,\ 3\rangle,\ \langle 1,\ 4\rangle,\ \langle 2,\ 3\rangle,\ \langle 2,\ 4\rangle\}$

Элементы множества можно преобразовывать. Выражение $\{P(x) : x \in S\}$ применяет P к каждому элементу:

$Q : \{x * x : x \in 1\ ..\ 3\}$

$A : \{1,\ 4,\ 9\}$

Полезным также бывает оператор выбора $CHOOSE\ x \in S : P(x)$, который выбирает случайный элемент множества S, удовлетворяющий предикату P. Стоит отметить, что проверщик моделей не будет выполнять полный перебор возможных вариантов выбора, а зафиксирует один.

Q : CHOOSE $x \in 1\ ..\ 100 : x > 42$

$A : 43$

Ожидаемо присутствуют следующие операторы:

$Q : \{\} \in \{\{\}\}$

A : TRUE

$Q : \{\} \notin \{\{\}\}$

A : FALSE

$Q : \{1, 2\} \subseteq \{1, 2, 3\}$

A : TRUE

$Q : \{1, 2\} \cup \{3\}$

$A : \{1, 2, 3\}$

$Q : \{1, 2\} \cap \{2, 3\}$

$A : \{2\}$

$Q : \{1, 2\} \setminus \{2, 3\}$

$A : \{1\}$

Q : SUBSET $\{1, 2\}$

$A : \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$

Q : UNION $\{\{1\}, \{2\}\}$

$A : \{1, 2\}$

2.3.3 Логика предикатов

Комбинация логики высказываний и теории множеств дает логику предикатов, которая позволяет писать утверждения об элементах множества и является базисом разработки на TLA$^+$. Логика предикатов добавляет кванторы существования ($\exists$) и всеобщности ($\forall$):

$Q : \exists\, x \in \{1, 2\} : x < 0$

A : FALSE

$Q : \forall\, x \in \{1, 2\} : x > 0$

A : TRUE

2.3.4 Моделирование данных

Помимо множеств для моделирования данных в TLA$^+$ используются *кортежи* и *структуры*.

Кортежи - упорядоченные последовательности элементов произвольного типа. Основные операторы над ними:

Q : $Head(\langle 0, \text{“A”}\rangle)$

A : 0

Q : $Tail(\langle 0, \text{“A”}\rangle)$

A : $\langle \text{“A”}\rangle$

Q : $Append(\langle 0\rangle, \text{“A”})$

A : $\langle 0, \text{“A”}\rangle$

Q : $\langle 0\rangle \circ \langle \text{“A”}\rangle$

A : $\langle 0, \text{“A”}\rangle$

Q : $Len(\langle 0, \text{“A”}, \text{TRUE}\rangle)$

A : 3

Q : DOMAIN $\langle \text{“One”}, \text{“Two”}\rangle$

A : $\{1, 2\}$

Структуры являются ассоциативными массивами и конструируются конутсрукциями вида $[k1 \mapsto v1, k2 \mapsto v2]$:

Q : $q \triangleq [x \mapsto \{\}, y \mapsto \{0, \text{“A”}\}]$

$q.y$

A : $\{0, \text{“A”}\}$

Q : DOMAIN q

A : $\{\text{“x”}, \text{“y”}\}$

Определен синтаксис генераторов структур:

Q : $[x : \{1\}, y : \{0, 1\}]$

A : $\{[x \mapsto 1, y \mapsto 0], [x \mapsto 1, y \mapsto 1]\}$

2.3.5 Операторы и функции

Для манипуляции данными используются *операторы* и *функции*.

Операторы TLA$^+$ похожи на функции в обычных языках программирования:

Q : $IsPrime(x) \triangleq x > 1 \land \neg\exists\, d \in 2\,..\,x-1 : x\%d = 0$

$IsPrime(37)$

A : TRUE

Функции же больше похожи на словари, где ключи лежат в некотором множестве определения, а значения в множестве значений:

Q : $Squares[x \in 1\,..\,4] \triangleq x * x$

$Squares[4]$

A : 16

Q : DOMAIN $Squares$

A : $\{1, 2, 3, 4\}$

На самом деле кортежи и структуры реализованы поверх функций. В случае кортежей областью определения являются натуральные чилса.

2.3.6 PlusCal

Для удобства разработчиков был создан язык PlusCal, код которого пишется в многострочных комментариях и транспилируется в TLA$^+$. Язык похож на C и предоставляет привычные выражения контроля, такие как *while*, *if-then-else*, *goto*. Полезено выражение $with\ x \in S$, которое позволяет указать проверщику моделей, что нужно перебрать все возможные значения x из множества S.

Пример типичного кода на PlusCal:

--algorithm $counter$

variables $x = 0$

begin

while $x < 5$ **do**

with $inc \in \{1, 2\}$ **do**

$x := x + inc$

end with ;

end while ;

end algorithm ;

2.3.7 Темпоральная логика

Выше уже упоминались операторы темпоральной логики. Они позволяют нам определять свойства *живости (анг. liveness)*, которые описывают, что же должно обязательно происходить в нашей системе со временем.

Попробуем запустить проверку примера выше на соответствие различным темпоральным свойствам и посмотрим на результат:

- $\Box(x > 0)$: x всегда больше нуля. Успех проверки.
- $\Diamond(x = 5)$: x обязательно будет равен 5. Поскольку наш счетчик уменьшается или на 1, или на 2, проверщик находит исполнение, в котором свойство не выполняется: $0, 2, 4, 6$. Свойство не выполняется.
- $(x = 0) \rightsquigarrow (x > 3)$: если значение x было 0, то обязательно будет больше 3. Успех проверки.

Как видно, знания базовых элементов математики вроде теории множеств почти достаточно для использования TLA$^+$. Новым для разработчиков может стать формализм темпоральной логики, однако и он прост в изучении, поскольку строится на интуитивном понимании событийности. Также язык предоставляет привычные разработчикам управляющие конструкции. Основываясь на сказанном, язык спецификаций TLA$^+$ составит основу разрабатываемой методики.

Методика

Foo bar

3.1 Языки и нотации

3.2 Уточнение требований

3.3 Шаблоны

3.3.1 Алгоритм

3.3.2 Машина состояний

3.4 Примеры

3.4.1 Денежный перевод

3.4.2 Светофоры

3.5 Анализ применимости

3.5.1 Стоимость

3.5.2 Достоинства

3.5.3 Ограничения

Доработка инструментов TLA+

4.1 Подсветка синтаксиса

4.2 Neovim плагин

Выводы

Литература

[1] Phil Koopman. A case study of toyota unintended acceleration and software safety. *Electrical And Computer Engineering*, 2014.

[2] Nancy G. Leveson. The therac-25: 30 years later. *IEEE Computer*, 2017.

[3] Tom Cargill. Extreme programming challenge fourteen. 2009.

[4] Kathleen Fisher et al. The hacms program: using formal methods to eliminate exploitable bugs. *Philos Trans A Math Phys Eng Sci*, 2017.

[5] Chris Newcombe. Why amazon chose tla+. *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25—-39, 2014.

[6] Bpmn reference. https://www.bpmn.org/.

[7] Uml reference. https://www.uml-diagrams.org/class-reference.html.

[8] y. x. 1000.

[9] y. x. 100.

[10] y. x. 1000.

[11] y. x. 1000.

[12] August-Wilhelm Scheer. Process modeling using event-driven process chains. 2005.

[13] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. 1936.

[14] Tla+ reference. https://lamport.azurewebsites.net/tla/tla.html.

[15] Alloy reference. https://alloytools.org/.