

# 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 обеспечения 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 инвариантов 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

<b>1</b>	<b>Введение</b>	<b>4</b>
1.1	Основные понятия . . . . .	4
1.2	Цели работы . . . . .	4
1.3	Актуальность работы . . . . .	5
1.4	Анализ подходов . . . . .	6
1.5	Структура работы . . . . .	7
<b>2</b>	<b>Выбор инструмента</b>	<b>8</b>
2.1	Сравнение технологий . . . . .	8
2.2	Моделирование времени . . . . .	9
2.3	Возможности TLA <sup>+</sup> . . . . .	10
2.3.1	Логика высказываний . . . . .	10
2.3.2	Теория множеств . . . . .	11
2.3.3	Логика предикатов . . . . .	12
2.3.4	Моделирование данных . . . . .	13
2.3.5	Операторы и функции . . . . .	14
2.3.6	PlusCal . . . . .	14
2.3.7	Темпоральная логика . . . . .	15
<b>3</b>	<b>Методика</b>	<b>16</b>
3.1	Языки и нотации . . . . .	16
3.2	Уточнение требований . . . . .	17

3.3	Спецификация . . . . .	17
3.3.1	Алгоритм . . . . .	18
3.3.2	Автомат . . . . .	20
3.4	Эксплуатация . . . . .	23
3.5	Примеры . . . . .	24
3.5.1	Денежный перевод . . . . .	24
3.5.2	Перекресток . . . . .	24
3.5.3	Запрос цены . . . . .	34
3.6	Анализ применимости . . . . .	34
3.6.1	Опыт . . . . .	34
3.6.2	Достоинства . . . . .	35
3.6.3	Ограничения . . . . .	35
<b>4</b>	<b>Доработка инструментов TLA<sup>+</sup></b>	<b>36</b>
4.1	Подсветка синтаксиса . . . . .	36
4.2	Neovim плагин . . . . .	36
<b>5</b>	<b>Выводы</b>	<b>37</b>
<b>6</b>	<b>Литература</b>	<b>38</b>

# 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 конкурентности 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 поддержке 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, *декларативный* 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 математические 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 Главе 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  $\text{TLA}^+$ . И, наконец, описаны концепты  $\text{TLA}^+$ , 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 верно”. 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  $Q$  ( $P \leadsto Q$ )

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<sup>+</sup>

Большинство математических концептов, используемых в TLA<sup>+</sup>, просты. Данная секция дает представление о том, как выглядит написание выражений в TLA<sup>+</sup>.

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

### 2.3.1 Логика высказываний

TLA<sup>+</sup> 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 “А верно И (В верно ИЛИ С ложно)“, а выражение

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

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

TLA<sup>+</sup> умеет вычислять высказывания:

$Q : \text{FALSE} \Rightarrow \text{FALSE}$

$A : \text{TRUE}$

$Q : \text{FALSE} \Rightarrow \text{FALSE}$

$A : \text{TRUE}$

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

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

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

### 2.3.2 Теория множеств

Следующим важным формализмом  $\text{TLA}^+$  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  $\text{CHOOSE } x \in S : P(x)$ , 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.

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

$A : 43$

Ожидаемо присутствуют следующие операторы:

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

$A : \text{TRUE}$

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

$A : \text{FALSE}$

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

$A : \text{TRUE}$

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

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

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

$A : \{2\}$

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

$A : \{1\}$

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

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

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

$A : \{1, 2\}$

### 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  $\text{TLA}^+$ . Логика предикатов добавляет кванторы существования ( $\exists$ ) и всеобщности ( $\forall$ ):

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

$A : \text{FALSE}$

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

$A : \text{TRUE}$

## 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:

$$Q : Head(\langle 0, "A" \rangle)$$

$$A : 0$$

$$Q : Tail(\langle 0, "A" \rangle)$$

$$A : \langle "A" \rangle$$

$$Q : Append(\langle 0 \rangle, "A")$$

$$A : \langle 0, "A" \rangle$$

$$Q : \langle 0 \rangle \circ \langle "A" \rangle$$

$$A : \langle 0, "A" \rangle$$

$$Q : Len(\langle 0, "A", TRUE \rangle)$$

$$A : 3$$

$$Q : DOMAIN \langle "One", "Two" \rangle$$

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

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

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

$$q.y$$

$$A : \{0, "A"\}$$

$$Q : DOMAIN q$$

$$A : \{"x", "y"\}$$

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<sup>+</sup> похожи на функции в обычных языках программирования:

$$Q : IsPrime(x) \triangleq x > 1 \wedge \neg \exists d \in 2 \dots x - 1 : x \% d = 0$$
$$IsPrime(37)$$

A : TRUE

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

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

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 числа.

### 2.3.6 PlusCal

Для удобства разработчиков был создан язык PlusCal, код которого пишется в многострочных комментариях и транпилируется в TLA<sup>+</sup>. Язык похож на C и предоставляет привычные выражения контроля, такие как *while*, *if-then-else*, *goto*. Полезно 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* (англ. *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.
- $(x = 0) \leadsto (x > 3) : \text{если значение } x \text{ было } 0, \text{ то обязательно будет больше } 3. \text{ Успех проверки.}$

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

В данной главе приводятся рассуждения на тему наиболее популярных вариантов специфицирования бизнес-процессов. Затем с опором на это разрабатывается методика специфицирования, состоящая из этапов уточнения требований, написания спецификации, ее эволюции со временем и поддержки. Наконец, приводятся примеры применения и анализ применимости, основанный на опыте использования.

## 3.1 Языки и нотации

От сложившихся процессов и состава сотрудников отдела разработки зависит и процесс спецификации. Проведем классификацию этого процесса по возрастанию комплексности подхода:

- Требования формулируются владельцем продукта. Зачастую в устном виде.
- Для формулирования привлекается бизнес-аналитика. Постановки записаны на естественных языках (русском, английском и т.д.). Определены мотивация, цели и критерии готовности (анг. DOD - definition of done).
- Для формулирования помимо бизнес-аналитики привлекается системная аналитика. Спецификации могут быть снабжены диаграммами в таких нотациях как, UML [7] и BPMN [7]. Конечно, существуют и другие нотации (например, такие как BPEL [16] и YAWL [17]), однако они не так распространены.

Используя формальные методы, мы насыщаем спецификацию *семантикой*, которая отсутствует как в естественных языках, так и в перечисленных нотациях. Однако, чтобы приступить к этому, нам необходимо лучше выяснить нужные бизнесу инварианты.

## 3.2 Уточнение требований

Рабочий способ “конденсации” инвариантов для нашей формальной спецификации — через общение с визионерами продукта. Эти люди хорошо понимают пользователей и цену ошибки. При выяснении требований полезно задавать следующие вопросы в контексте специфицируемого процесса:

- Когда можно считать процесс успешно завершённым?
- Что может нанести ущерб клиенту?
- Из-за чего клиент может испытать раздражение?
- Что может принести нам финансовые убытки?
- Что может нанести нам репутационный ущерб?
- Есть ли требования от регулятора?

Ответы на эти вопросы помогают понять, что важно бизнесу и клиенту, чего происходить не должно и что обязательно должно произойти. Собрав такие свойства, мы подходим непосредственно к этапу формальной спецификации.

## 3.3 Спецификация

Одним из преимуществ того, что TLA<sup>+</sup> — язык спецификации, является то, что его операторы могут обладать намного большей выразительной силой, чем функции в обычных программах. Это также является и недостатком: если спецификация использует “слишком выразительный” оператор, она не может быть напрямую транслирована в код. Обычно это нормально: разрабатывая код для большой системы мы вряд беспокоимся о том, корректна ли наша функция

сортировки.

Чтобы процесс спецификации с применением  $TLA^+$  был доступен для повседневного применения, нам необходимо исследовать возможные подходы, “строительные блоки“, от которых можно отталкиваться.

### 3.3.1 Алгоритм

Выделим первый класс процессов: *алгоритмы*. Под алгоритмом будем подразумевать процессы, которые должны завершаться и выдавать результат (в отличие от процессов, которые идут бесконечно и непрерывно взаимодействуют с окружением).

В свою очередь алгоритмы можно разделить на *однопроцессные* и *многопроцессные*. В контексте алгоритма под процессом понимается актор алгоритма, уместна аналогия с системными процессами.

#### Однопроцессный

Большинство однопроцессных алгоритмов можно свести к описанию по следующему шаблону:

```
--algorithm singleProcess
variables
  input = ...
  result ;
define
  Expected(input)  $\triangleq$  ...
end define ;
begin
  // имплементация
  assert output = Expected(input) ;
end algorithm ;
```

Оператор *Expected* — то, что мы на самом деле пытаемся реализовать: он берет некоторые входные данные и возвращает значение, которое должен производить алгоритм, если он корректен. Помимо него в блоке *define* могут быть определены любые операторы, которые могут использоваться алгоритмом и которые мы не планируем здесь верифицировать. Далее следует код алгоритма на PlusCal, который мы хотим проверить на работоспособность на заданном множестве входных данных. Алгоритм сохраняет результат своей работы в переменную *output*, которой мы намеренно не дали начального значения. Также в секцию *variables* должны быть помещены любые переменные, нужные алгоритму, поскольку в теле алгоритма новые объявления невозможны. В конце происходит проверка на соответствие вывода алгоритма и ожидаемого результата.

Безусловно, шаблон не является строгим. Если алгоритм сложен, мы можем выносить новые процедуры и макросы, чтобы разбить его на части.

Спецификации однопроцессных алгоритмов могут напоминать *тестирование на основе модели* (анг. *property-based testing*). Однако, здесь имеется отличие в уровне абстракции. В то время как тестирование применяется к конкретному коду, в спецификации мы можем этот уровень выбрать по нашему усмотрению.

## Многопроцессный

Многопроцессные алгоритмы похожи на однопроцессные, кроме того что в них мы также хотим проверять, что все процессы завершатся. Вместо прямого кодирования выражения *assert* мы запишем это как свойство живучести (анг. *liveness*). Это означает использование оператора “в конце концов всегда” ( $\Diamond\Box$ ), который проверяет, что алгоритм завершается с некоторым истинным условием в конце.

Шаблон для таких алгоритмов выглядит следующим образом:

```
--algorithm singleProcess
```

**variables**

*input* = ...

**define**

*Success*  $\triangleq$   $\Diamond\Box(\textit{someProperty})$

**end define ;**

**fair process** *proc*  $\in 1 \dots N$

**variables** ... ;

**begin**

// имплементация

**end process ;**

**end algorithm ;**

Процессы в PlusCal можно моделировать с помощью специальной конструкции *process*. У каждого процесса должен быть свой индивидуальный идентификатор, который присваивается в блоке  $\in 1..N$ . Для запуска необходимо указать проверщику, что формула *Success* это свойство, которое необходимо проверять.

Надо сказать, что в чистом виде этих шаблонов обычно недостаточно для написания спецификации. Однако вместе со следующим подходом они дают хороший результат.

### 3.3.2 Автомат

Как упоминалось выше, спецификации выразительнее, чем код, и за это есть своя цена: не всегда очевидно, как перейти от спецификации к реальности. Решением этой проблемы является подход, в котором мы пишем максимально абстрактную спецификацию и наполняем ее деталями в нужных местах, которые ближе к реализации в реальном коде. Для этого можно использовать паттерн *автомата* (англ. *state machine*).

Автомат — система с конечным множеством внутренних “состояний” наряду

с множеством переходов между этими состояниями. Переходы могут инициироваться внешними событиями.

Шаблон для кодирования автомата выглядит так:

**--algorithm** *stateMachine*

**variables**

*flag1* = FALSE,

...

*flagN* = FALSE ;

**begin**

*Action*:

**either**

**await** *flag1* ;

...

**or**

**await**  $\neg$ *flag1* ;

...

**or**

**await** *flagN*

...

**or**

**await**  $\neg$ *flagN* ;

...

**end either** ;

**goto** *Action* ;

**end algorithm** ;

Для знакомства с шаблоном опробуем его на небольшом примере. Пусть мы определяем поведение светофора. Светофор может гореть одним из цветов: зеленый, желтый, красный.

**--algorithm** *trafficLight*

**variables**

*red* = TRUE,

*yellow* = FALSE,

*green* = FALSE,

**begin**

*Action*:

**either**

**await** *green* ;

*yellow* = TRUE ;

**or**

**await** *yellow* ;

*red* = TRUE ;

**or**

**await** *red* ;

*yellow* = TRUE ;

**end either** ;

**goto** *Action* ;

**end algorithm** ;

Каких свойств можно потребовать от светофора? Например, такое: если горит красный, то когда-нибудь загорится и зеленый (и наоборот). На языке TLA<sup>+</sup> оно формулируется как  $red \leadsto green \wedge green \leadsto red$ . Запускаем...спецификация падает. Проверщик находит исполнение, в котором красный лишь чередуется с желтым. В логике желтого цвета ошибка: он переключается только с желтого на красный. Добавим новую переменную *moveTo*, которая будет отвечать за направление переключения цветов в данный момент и поправим логику алгоритма:

**--algorithm** *trafficLight*

**variables**

*moveTo* = "green",

```

red = TRUE,
yellow = FALSE,
green = FALSE,
begin
  Action:
    either
      await green ;
      yellow := TRUE ;
      moveTo := "red"
    or
      await yellow ∧ moveTo = "red" ;
      red := TRUE ;
    or
      await yellow ∧ moveTo = "green" ;
      green := TRUE ;
    or
      await red ;
      yellow := TRUE ;
      moveTo := "green"
    end either ;
  goto Action ;
end algorithm ;

```

Теперь спецификация проходит: красный всегда сменяется зеленым и наоборот.

## 3.4 Эксплуатация

После того как написана спецификация и бизнес-процесс, наступает этап эксплуатации. Формальную спецификацию не стоит воспринимать как однора-



зовый артефакт. Она является отличным источником документации и маркером слабых мест бизнес-процесса.

Текущие инструменты не позволяют из спецификации сгенерировать рабочий код на произвольном языке программирования и наоборот: это горячее направление для исследования для всех систем формальной верификации. Чтобы спецификации не устаревали, их необходимо дорабатывать при изменениях бизнес-процесса, как и любую другую документацию. Это одно из ограничений подхода.

## 3.5 Примеры

В данной секции рассмотрим применение разработанного метода на нескольких примерах. Целью является демонстрация рассуждений и мыслительного процесса спецификации в зависимости от формата исходной постановки задачи.

### 3.5.1 Денежный перевод

### 3.5.2 Перекресток

Рассмотрим пример задачи с высокоуровневой постановкой, которая может исходить, например, от владельцев продукта.

**Задача.** Есть нерегулируемый перекресток двух равнозначных дорог, он должен быть безопасен.

Другими словами, требуется разработать систему правил проезда перекрестка, при соблюдении которых гарантированно никто не пострадает.

При пересечении перекрестка у машин есть 4 возможных направления движения: поехать прямо, повернуть налево, повернуть направо, развернуться. Для удобства обозначим ответвления дороги как *N*, *E*, *S*, *W*. Сначала смоделируем сам перекресток без каких-либо правил проезда. Интуиция подсказывает, что

состояния машины на перекрестке (подъезжает к нему, стоит перед его началом, едет по нему) можно попробовать смоделировать в виде автомата. Для этого воспользуемся нашим шаблоном:

```

MODULE crossroads
EXTENDS Sequences, Integers, TLC, FiniteSets, Helpers
CONSTANTS Cars

DirsSeq  $\triangleq$   $\langle \text{"N"}, \text{"E"}, \text{"S"}, \text{"W"} \rangle$ 
Dirs  $\triangleq$  Range(DirsSeq)
--algorithm crossroadPass
variables
    queue = [ $d \in \textit{Dirs} \mapsto \langle \rangle$ ],

fair process car  $\in$  Cars
variables
    from  $\in$  Dirs, to  $\in$  Dirs, state = "Initial" ;
begin
    Action:
        either
            await state = "Initial" ;
            state := "Waiting" ;
            queue[from] := Append(queue[from], self) ;
        or
            await state = "Waiting" ;
            state := "Passing" ;
        or
            await state = "Passing" ;
            state := "Initial" ;
            queue[from] := Tail(queue[from]) ;
        end either ;

```

**goto** *Action* ;

**end process** ;

**end algorithm** ;

Направления движения моделируем и как множество *Dirs*, и как последовательность *DirsSeq*. В то время как множество полезно для ветвления алгоритма по всем направлениям, последовательность поможет устанавливать относительность направлений (например “Проезд из S в E — поворот направо”).

Подъезжая к перекрестку, машина становится в очередь *queue* на проезд. Чтобы смоделировать непрерывный поток машин по некоторому направлению, после окончания алгоритма делаем *goto* в начальное состояние. Ключевое слово *fair* перед *process* подразумевает, что машины не перестанут двигаться по алгоритму, пока будут доступные шаги.

Пора определить, что такое безопасный перекресток. Будем считать, что перекресток безопасен, если для любой пары машин верно: когда они переезжают перекресток одновременно, их пути не пересекаются, то есть они не могут столкнуться. Рассмотрение всех возможных вариантов столкновения мы пока отложим и остановимся на простом подмножестве: машины могут столкнуться, если они едут в одну и ту же полосу.

$$MkCar(f, t, s) \triangleq [from \mapsto f, to \mapsto t, state \mapsto s]$$

$$\begin{aligned} Crashing(car1, car2) &\triangleq \\ &\wedge car1.state = \text{“Passing”} \\ &\wedge car2.state = \text{“Passing”} \\ &\wedge car1.to = car2.to \end{aligned}$$

$$\begin{aligned} NoCrash &\triangleq \\ &\forall c1, c2 \in Cars : \\ &(c1 \neq c2) \Rightarrow \neg Crashing( \\ &\quad MkCar(from[c1], to[c1], state[c1]), \\ &\quad MkCar(from[c2], to[c2], state[c2])) \end{aligned}$$

)

Пока будем запускать модель для двух потоков машин  $c1$  и  $c2$ . Просим проверщик моделей запускаться с инвариантом  $NoCrash$  и закономерно получаем историю, которая ему противоречит: к примеру, движения  $S \rightarrow N$  и  $W \rightarrow E$  могут привести к столкновению.

Перед выездом на перекресток добавим проверку  $CanMove$ , которая реализует известное “правило правой руки”: нужно уступать дорогу тем, кто справа от нас и чья траектория пересекается с нашей.

**algorithm** *crossroadPass*

**variables**

$queue = [d \in Dirs \mapsto \langle \rangle],$   
 $wantTo = [d \in Dirs \mapsto \{\}],$   
 $wantFrom = [d \in Dirs \mapsto \{\}],$   
 $passing = [d \in Dirs \mapsto \{\}];$

**define**

$CanMove(car, from, to) \triangleq$

LET

$Candidates \triangleq Heads(queue)$

$To(t) \triangleq wantTo[t] \cap Candidates$

$From(f) \triangleq wantFrom[f] \cap Candidates$

$Reversing(d) \triangleq To(d) \cap From(d)$

$Conflicts(f, t) \triangleq ((To(t) \cap From(RightTo[f])) \setminus Reversing(t)) \cup passing[t]$

$Reversal(f, t) \triangleq (f = t) \Rightarrow Cardinality(To(t)) = 1$  reversing car itself

IN

$car \in Candidates \wedge$

$Cardinality(Conflicts(from, to)) = 0 \wedge$

$Reversal(from, to)$

**end define ;**

**fair process**  $car \in Cars$

**variables**

$from \in Dirs, to \in Dirs, state = \text{"Initial"};$

**begin**

*Action:*

**either**

**await**  $state = \text{"Initial"};$

$state := \text{"Waiting"};$

$queue[from] := Append(queue[from], self);$

$wantTo[to] := wantTo[to] \cup \{self\};$

$wantFrom[from] := wantFrom[from] \cup \{self\};$

**or**

**await**  $state = \text{"Waiting"};$

**await**  $CanMove(self, from, to);$

$state := \text{"Passing"};$

$passing[to] := passing[to] \cup \{self\};$

**or**

**await**  $state = \text{"Passing"};$

$state := \text{"Initial"};$

$queue[from] := Tail(queue[from]);$

$wantTo[to] := wantTo[to] \setminus \{self\};$

$wantFrom[from] := wantFrom[from] \setminus \{self\};$

$passing[to] := passing[to] \setminus \{self\};$

**end either ;**

**goto** *Action* ;

**end process ;**

**end algorithm ;**

Такая спецификация удовлетворяет инварианту *NoCrash!* Помимо безопас-

ности естественным свойством перекрестка видится гарантия того, что каждая машина рано или поздно проедет, куда ей нужно. Сформулируем это свойство с помощью темпорального оператора “когда-нибудь”( $\Diamond$ ).

$$EveryonePasses \triangleq$$

$$\forall c \in Cars : \Diamond(state[c] = \text{“Passing”})$$

Ошибка! Запуск проверки обнаружил историю, в которой машина может бесконечно долго стоять в ожидании своей очереди (Пусть  $c1 : S \rightarrow S$  и  $c2 : N \rightarrow S$ . Если поток машин  $c2$  непрерывен, машина  $c1$  будет обречена на вечное ожидание возможности разворота). Очевидно, в реальной жизни не может существовать бесконечного потока машин, однако полученный результат свидетельствует о ситуации на дороге, которую стоит учитывать при проектировании развязок.

Вернемся к нашему упрощенному определению столкновения машин и расширим его до всех возможных ситуаций:

$$IndOf(dir) \triangleq Matching(DirsSeq, dir)$$

$$RightTo[x \in Dirs] \triangleq$$

$$LET RightInd \triangleq (IndOf(x) \% 4) + 1$$

$$IN DirsSeq[RightInd]$$

$$OppTo[x \in Dirs] \triangleq$$

$$LET OppInd \triangleq ((IndOf(x) + 1) \% 4) + 1$$

$$IN DirsSeq[OppInd]$$

$$LeftTo[x \in Dirs] \triangleq$$

$$LET LeftInd \triangleq ((IndOf(x) + 2) \% 4) + 1$$

$$IN DirsSeq[LeftInd]$$

$$MkCar(f, t, s) \triangleq [from \mapsto f, to \mapsto t, state \mapsto s]$$

$$Straight(car) \triangleq Abs(IndOf(car.from) - IndOf(car.to)) = 2$$

$$Left(car) \triangleq Abs(IndOf(car.from) - IndOf(car.to)) = 3$$

$$Reverse(car) \triangleq car.from = car.to$$

$$\begin{aligned}
Crashing(car1, car2) &\triangleq \\
&\wedge car1.state = \text{"Passing"} \\
&\wedge car2.state = \text{"Passing"} \\
&\wedge \vee car1.to = car2.to \\
&\quad \vee \wedge Straight(car1) \\
&\quad \quad \wedge \vee car2.from = RightTo[car1.from] \\
&\quad \quad \quad \vee Straight(car2) \\
&\vee \wedge Left(car1) \\
&\quad \wedge \vee car2.from = RightTo[car1.from] \\
&\quad \quad \vee car2.from = OppTo[car1.from] \\
&\quad \quad \vee \wedge car2.from = LeftTo[car1.from] \\
&\quad \quad \quad \wedge Left(car2) \vee Straight(car2) \\
&\vee \wedge Reverse(car1)
\end{aligned}$$

Обновленное условие столкновения дает интересные результат: правило правой руки не помогает в разрешении ситуации  $c1 : S \rightarrow S, c2 : N \rightarrow N$ , то есть когда две машины едут навстречу друг другу и обе хотят развернуться.

Итоговая спецификация:

<b>MODULE</b> <i>crossroads</i>
<b>EXTENDS</b> <i>Sequences, Integers, TLC, FiniteSets, Helpers</i>
<b>CONSTANTS</b> <i>Cars</i>
$DirsSeq \triangleq \langle \text{"N"}, \text{"E"}, \text{"S"}, \text{"W"} \rangle$
$IndOf(dir) \triangleq Matching(DirsSeq, dir)$
$Dirs \triangleq Range(DirsSeq)$
$RightTo[x \in Dirs] \triangleq$
$\quad \text{LET } RightInd \triangleq (IndOf(x)\%4) + 1$
$\quad \text{IN } DirsSeq[RightInd]$
$OppTo[x \in Dirs] \triangleq$
$\quad \text{LET } OppInd \triangleq ((IndOf(x) + 1)\%4) + 1$

$\text{IN } \text{DirsSeq}[\text{OppInd}]$   
 $\text{LeftTo}[x \in \text{Dirs}] \triangleq$   
 $\text{LET } \text{LeftInd} \triangleq ((\text{IndOf}(x) + 2) \% 4) + 1$   
 $\text{IN } \text{DirsSeq}[\text{LeftInd}]$   
 $\text{MkCar}(f, t, s) \triangleq [\text{from} \mapsto f, \text{to} \mapsto t, \text{state} \mapsto s]$   
 $\text{Straight}(\text{car}) \triangleq \text{Abs}(\text{IndOf}(\text{car}.\text{from}) - \text{IndOf}(\text{car}.\text{to})) = 2$   
 $\text{Left}(\text{car}) \triangleq \text{Abs}(\text{IndOf}(\text{car}.\text{from}) - \text{IndOf}(\text{car}.\text{to})) = 3$   
 $\text{Reverse}(\text{car}) \triangleq \text{car}.\text{from} = \text{car}.\text{to}$   
 $\text{Crashing}(\text{car1}, \text{car2}) \triangleq$   
 $\wedge \text{car1}.\text{state} = \text{"Passing"}$   
 $\wedge \text{car2}.\text{state} = \text{"Passing"}$   
 $\wedge \vee \text{car1}.\text{to} = \text{car2}.\text{to}$   
 $\vee \wedge \text{Straight}(\text{car1})$   
 $\wedge \vee \text{car2}.\text{from} = \text{RightTo}[\text{car1}.\text{from}]$   
 $\vee \text{Straight}(\text{car2})$   
 $\vee \wedge \text{Left}(\text{car1})$   
 $\wedge \vee \text{car2}.\text{from} = \text{RightTo}[\text{car1}.\text{from}]$   
 $\vee \text{car2}.\text{from} = \text{OppTo}[\text{car1}.\text{from}]$   
 $\vee \wedge \text{car2}.\text{from} = \text{LeftTo}[\text{car1}.\text{from}]$   
 $\wedge \text{Left}(\text{car2}) \vee \text{Straight}(\text{car2})$   
 $\vee \wedge \text{Reverse}(\text{car1})$

**--algorithm** *crossroadPass*

**variables**

$\text{queue} = [d \in \text{Dirs} \mapsto \langle \rangle],$   
 $\text{wantTo} = [d \in \text{Dirs} \mapsto \{\}],$   
 $\text{wantFrom} = [d \in \text{Dirs} \mapsto \{\}],$   
 $\text{passing} = [d \in \text{Dirs} \mapsto \{\}];$

**define**



$CanMove(car, from, to) \triangleq$

**LET**

$Candidates \triangleq Heads(queue)$

$To(t) \triangleq wantTo[t] \cap Candidates$

$From(f) \triangleq wantFrom[f] \cap Candidates$

$Reversing(d) \triangleq To(d) \cap From(d)$

$Conflicts(f, t) \triangleq ((To(t) \cap From(RightTo[f])) \setminus Reversing(t)) \cup passing[t]$

$Reversal(f, t) \triangleq (f = t) \Rightarrow Cardinality(To(t)) = 1 \text{ reversing car itself}$

**IN**

$car \in Candidates \wedge$

$Cardinality(Conflicts(from, to)) = 0 \wedge$

$Reversal(from, to)$

**end define ;**

**fair process**  $car \in Cars$

**variables**

$from \in Dirs, to \in Dirs, state = \text{"Initial"};$

**begin**

*Action:*

**either**

**await**  $state = \text{"Initial"};$

$state := \text{"Waiting"};$

$queue[from] := Append(queue[from], self);$

$wantTo[to] := wantTo[to] \cup \{self\};$

$wantFrom[from] := wantFrom[from] \cup \{self\};$

**or**

**await**  $state = \text{"Waiting"};$

**await**  $CanMove(self, from, to);$

$state := \text{"Passing"};$

$passing[to] := passing[to] \cup \{self\};$

**or**

**await**  $state = \text{"Passing"};$

$state := \text{"Initial"};$

$queue[from] := Tail(queue[from]);$

$wantTo[to] := wantTo[to] \setminus \{self\};$

$wantFrom[from] := wantFrom[from] \setminus \{self\};$

$passing[to] := passing[to] \setminus \{self\};$

**end either ;**

**goto** *Action* ;

**end process ;**

**end algorithm ;**

Еще один интересный результат — обнаружение дедлока в ситуации, когда с разных сторон к перекрестку подъезжают сразу 4 машины с пересекающимися направлениями движения.

В этом примере мы начали со спецификации в одно предложение и пришли к необходимости правил по проезду перекрестка. За моделировав “правило правой руки”, выяснили часть его фундаментальных проблем:

- При неудачном стечении обстоятельств машины могут никогда не проехать перекресток.
- Порядок проезда не определен для встречных разворачивающихся машин.
- Порядок проезда не определен для 4 машин на разных концах перекрестка.

Хоть по отдельности такие ситуации могут происходить редко, каждая из них может провоцировать аварийную ситуацию. Поэтому полезно знать о подобных ограничениях при проектировании городских улиц.

### 3.5.3 Запрос цены

## 3.6 Анализ применимости

Этот раздел завершает рассказ о методике, приводя рассуждения о накладных расходах ее применения, ее достоинствах и ограничениях. Все рассуждения выведены эмпирическим путем.

### 3.6.1 Опыт

Был проведен эксперимент: методика была опробована на практике. Несколько разработчикам было выделено время на изучение TLA<sup>+</sup> и интеграцию в повседневную работу. От старта эксперимента до получения первых полезных спецификаций у разных участников ушло от нескольких дней до двух недель. Разработчики отметили простоту концептов, выразительность языка и возможность переиспользования их опыта программирования.

Наиболее сложным оказалось формулирование темпоральных свойств. В связи с этим был разработан описанный выше набор вопросов, облегчающих обнаружение инвариантов. Также выяснилось, что качество инструментов для TLA<sup>+</sup> уступает качеству оных для языков программирования. Это послужило мотивацией для ряда разработок, о которых речь пойдет в последней главе работы.

Полученные спецификации помогли выявить изъяны в исходном дизайне и подсветить хрупкие места системы. Такие места — естественный кандидат для мониторинга. Плюс мониторинга, выведенного из спецификации в возможности отслеживания свойства, сформулированных на языке, понятном не только разработчикам.

По итогу эксперимента стали ясны достоинства и ограничения методики, опишем их детальнее.

### 3.6.2 Достоинства

- Проверщик моделей хорошо находят ошибки, проявляющиеся в конкурентном окружении с неопределенным порядком событий.
- Процесс спецификации выявляет противоречивость в требованиях и возможную недостижимость (или ограниченную достижимость) желаемых свойств.
- Спецификации являются полезным дополнением к документации бизнес-процессов.
- Язык TLA<sup>+</sup> построен на простых математических концептах (теория множеств, логика предикатов), знакомых многим разработчикам. Это делает его доступным для изучения в отличие от интерактивных доказателей теорем (таких как Coq, Isabelle, и т.д.).
- Неожиданным стало наблюдение о том, что накопление опыта формальной верификации смещает парадигму мышления и помогает видеть больше ошибок даже до начала применения методики.

### 3.6.3 Ограничения

# **Доработка инструментов TLA<sup>+</sup>**

## **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/>.
- [16] Bpel reference. [http://docs.openlinksw.com/virtuoso/bpel\\_reference/](http://docs.openlinksw.com/virtuoso/bpel_reference/).
- [17] Yawl reference. <https://yawlfoundation.github.io/assets/files/YAWLTechnicalManual4.pdf>