

# Техническая документация PLCopen

рабочая группа

## Создание руководств по разработке ПО

представляет

## Руководство по языку SFC

версия 1.0, официальный релиз

### Отказ от ответственности

Название 'PLCopen<sup>®</sup>' является зарегистрированным товарным знаком и совместно с логотипом PLCopen является собственностью ассоциации PLCopen.

Данный документ предоставляется "как есть" и в будущем может быть подвергнут изменениям и исправлениям. PLCopen не предоставляет никакие гарантии (явные или подразумеваемые), включая любые гарантии по поводу пригодности использования документа для конкретной цели. Ни при каких обстоятельствах PLCopen не несет ответственности за ущерб или убытки, вызванные ошибками в данном документе или его использованием.

Copyright © 2018 by PLCopen. Все права защищены

Дата публикации: 03.06.2018

Переводчик:	Е.Кислов
Редактор:	А.Осинский
Версия перевода:	1.0
Дата публикации:	07.09.2018

---

## Оглавление

Оглавление.....	2
Список авторов и история версий.....	4
Глоссарий .....	5
1. Введение .....	6
1.1. Роль SFC в модульности ПО .....	7
2. Основы SFC.....	9
2.1. Шаги и переходы .....	9
2.2. Действия .....	11
2.3. Классификаторы действий.....	11
2.4. Обработка SFC-схемы .....	12
2.5. Расходимость и сходимость .....	12
3. Структурные свойства SFC.....	15
3.1. Структура процесса управления.....	15
3.2. Параллельные процессы .....	19
3.3. Параллельность в SFC-схеме .....	20
3.4. Действия .....	21
3.4.1. Вступление .....	21
3.5. Классификаторы действий.....	22
3.5.1. Вступление .....	22
3.6. Управление действиями .....	32
3.6.1. Функциональный блок ACTION_CONTROL.....	32
3.6.2. Последнее сканирование (final scan).....	33
4. Рекомендации по использованию SFC .....	37
4.1. Расходимость и сходимость .....	37
4.2. Линеаризация в SFC.....	41
4.3. Условия перехода .....	42
4.4. Не используйте пересекающиеся условия .....	45
4.5. Зависимость от предыдущего состояния .....	46
4.6. Параллельные ветви .....	48
4.7. Независимость действий .....	51
4.8. Классификаторы S и R .....	53
4.9. Флаги и неявные переменные шагов .....	54

---

4.10. Независимость действий .....	54
5. Диаграммы состояний .....	55
6. Примеры использования диаграмм состояний.....	60
6.1. Простой пример управления двигателем .....	60
6.1.1. Основная информация.....	60
6.1.2. Диаграмма состояний .....	60
6.1.3. Реализация на SFC .....	61
6.2. Расширенный пример управления двигателем .....	62
6.2.1. Основная информация.....	62
6.2.2. Диаграмма состояний .....	62
6.2.3. Реализация на SFC .....	63
6.3. Пример перевода диаграммы RackML в SFC-схему .....	64
6.3.1. Основная информация о RackML .....	64
6.3.2. Преобразование диаграммы состояний в SFC.....	65
6.3.3. Обработка ошибок .....	66
6.3.4. Аспекты безопасности для различных режимов работы .....	69
6.3.5. Связь SFC и сетей Петри .....	71
6.3.6. Связь SFC и автоматов Мура и Мили .....	71

## Список авторов и история версий

Данный документ является официальным документом организации **PLCopen**:

### Руководство по языку SFC

Он основан на книге *Flavio Bonfatti, Paola Daniela Monari, Umberto Sampieri. IEC 61131-3 programming methodology*, представляет собой результат работы комитета **Язык SFC** рабочей группы **Создание руководств по разработке ПО** и включает вклад каждого из участников:

Участник	Компания
Bert van der Linden	<a href="#">ATS International</a>
Bernhard Werner	<a href="#">3S / Codesys</a>
Barry Butcher	<a href="#">Omron</a>
Hiroshi Yoshida	<a href="#">Omron</a>
Andreas Weichelt	<a href="#">Phoenix Contact</a>
Arnulf Meister	<a href="#">Phoenix Contact</a>
Eelco van der Wal	<a href="#">PLCopen</a>

### История версий

Версия	Дата	Описание
0.1	11.09.2014	Первая версия на базе материала Mr. Bonfatti (и других) с редактурой Eelco van der Wal.
0.2	02.10.2014	Комментарии от Barry Butcher. Версия представлена комитету.
0.3	28.11.2014	Комментарии от Hiroshi Yoshida и Andreas Weichelt.
0.4	03.12.2014	Дополнения по результатам встречи в Франкфурте.
0.5	03.2017	Дополнения по результатам встречи с Bert van der Linden в январе и добавление информации о RackML.
0.6	29.06.2017	Дополнения по результатам видеоконференций.
0.6a	31.07.2017	Дополнения по результатам видеоконференции.
0.7	24.08.2017	Дополнения по результатам видеоконференции. Добавлен п. 3.6.2.
0.8	02.2018	Дополнения по результатам видеоконференции.
0.99	24.02.2018	Предварительная версия для сбора комментариев.
1.0	03.06.2018	Официальный релиз.

---

## Глоссарий

**FBD** (*Function Block Diagram, язык функциональных блоков*) – графический язык программирования ПЛК.

**IL** (*Instruction List, список инструкций*) – текстовый язык программирования ПЛК, напоминающий язык Ассемблера.

**LD** (*Ladder Diagram, язык релейных схем*) – графический язык программирования ПЛК.

**SFC** (*Sequential Function Chart, язык последовательных схем*) – высокоуровневый графический язык программирования ПЛК, представляющий приложение в виде конечного автомата.

**ST** (*Structured Text, структурированный текст*) – высокоуровневый текстовый язык программирования ПЛК, напоминающий Паскаль.

**SFC-схема** – набор связанных шагов и переходов, описывающих структуру РОУ.

**РОУ** – обобщенное название функциональных блоков и программ.

**Ветвь** – фрагмент SFC-схемы без расходимости и сходимости.

**Действие** – операция, выполняемая на заданном шаге технологического процесса.

**Пересекающиеся условия** – условия, которые могут быть выполнять одновременно. Например, условия  $v1 > 10$  и  $v1 < 100$  являются пересекающимися. Рекомендуется избегать пересекающихся условий при определении переходов SFC-схемы.

**Переход** – процесс переключения активности шагов SFC-схемы при выполнении заданного условия.

**Проход** – процесс однократной обработки схемы системой исполнения ПЛК.

**Расходимость и сходимость** – способы представления альтернативных и параллельных ветвей.

**ФБ** – функциональный блок.

**Шаг** – элемент SFC-схемы, используемый для описания стадии технологического процесса.

## 1. Введение

Данный документ посвящен языку последовательных схем (Sequential Function Chart, SFC). SFC – это очень выразительный графический язык, являющийся частью стандарта [МЭК 61131-3](#). Важно отметить, что SFC является средством проектирования, а не программирования; он помогает создать структуру программы, но реализация все равно будет производиться на других языках.

SFC позволяет разбить пользовательскую программу на набор шагов, связанных между собой переходами. Каждый шаг включает в себя набор заданных действий. Каждый переход описывается условием перехода. Поскольку SFC подразумевает хранение данных (например, информации о текущем шаге), то он не может применяться для создания функций, так как они не имеют внутренней памяти. По этой причине язык SFC применяется только для проектирования структуры функциональных блоков и программ.

Если необходимо описать на SFC фрагмент программы (или ФБ), то вся программа (ФБ) также будет представлена на SFC. В рамках SFC-схемы компонент (POU), написанный на другом языке, можно представить в виде одного SFC-действия, которое выполняется под контролем вызывающего его объекта.

SFC является оптимальным решением для описания внутренней структуры POU при автоматизации последовательного процесса управления по следующим причинам:

- Если процесс состоит из нескольких последовательно выполняемых стадий, (например, процесс сборки изделия), то каждая стадия легко представляется в виде SFC-шага;
- Если процесс можно представить в виде конечного автомата, то состояния этого автомата представляются SFC-шагами, а переключения между ними – условиями перехода;
- SFC позволяет выполнить декомпозицию программы, что улучшит ее внутреннюю структуру и упростит отладку. При этом общий алгоритм выполнения будет наглядно визуализирован.

## 1.1. Роль SFC в модульности ПО

Необходимым условием увеличения эффективности и повышения качества ПО является использование модульного подхода. Концепция модульности подразумевает, что программа должна быть разбита на изолированные, слабо связанные блоки. Каждый из этих блоков может разрабатываться и тестироваться отдельно от остальных.

Для использования модульного подхода при программировании ПЛК процесс разработки должен быть организован по определенной методологии. На наш взгляд, следует выделить два основных тезиса:

- *Сконцентрируйтесь на проектировании.* Перед началом кодирования алгоритмы и структура программы должны быть продуманы как можно тщательнее;
- *Сконцентрируйтесь на стандарте.* Используйте возможности МЭК 61131-3 при проектировании и кодировании ПО – они охватывают большинство потребностей, возникающих при программировании ПЛК.

Язык SFC является отличным выбором для использования на ранних (проектирование) и последующих стадиях разработки ПО для ПЛК. Это обусловлено следующими причинами:

- *Выразительная мощь.* Язык SFC обладает наглядностью диаграмм состояний и имеет общие черты с сетями Петри. Диаграммы состояний и сети Петри являются одними из наиболее подходящих средств для моделирования динамических систем и широко используются в различных отраслях. Поэтому SFC хорошо подходит для описания алгоритма программы;
- *Графическое представление процессов.* SFC – не единственный графический язык стандарта МЭК 61131-3, но в отличие от двух остальных (LD и FBD) он позволяет действовать на высоком уровне абстракции. Графическая основа делает язык простым в изучении и использовании, а также позволяет наглядно описывать последовательные алгоритмы на различных уровнях детализации;
- *Возможность предварительного проектирования.* Язык SFC можно начать использовать уже на самых ранних стадиях проекта, чтобы в общих чертах описать предполагаемое поведение системы управления. Это делает его хорошим инструментом для начального анализа, когда многие аспекты процесса еще не известны. Использование SFC не добавляет двусмысленности, свойственной естественным языкам. Это позволяет избежать недопонимания между заказчиком, инженером-технологом и программистом;
- *Возможность детализации.* SFC-схема, созданная на ранней стадии проектирования, по мере появления новой информации может дополняться и изменяться. Таким образом формируется требуемый уровень детализации процесса;
- *Естественная связь с другими языками.* Язык SFC является средством проектирования. Реализация алгоритмов (т.е. описание конкретных операций, производимых ПЛК) должна быть выполнена с помощью языков программирования. Возможность свободного выбора наиболее подходящего языка реализации повышает эффективность разработки и улучшает качество ПО;
- *Поддержка фрагментации кода.* Использование SFC позволяет разделить код на отдельные блоки, которые будут выполняться в разных циклах ПЛК. Таким образом, максимальное время цикла будет уменьшено. SFC-схема позволяет в явном виде выделить отдельные фрагменты программы и описать условия перехода между ними.

Создание SFC-схемы включает следующие этапы:

- Каждая стадия процесса представляется в виде *шага*;
- *Переходы* между шагами описываются с помощью *условий*;
- Операции, которые должны быть выполнены на том или ином шаге, кодируются в *действиях* этого шага. Каждое действие имеет *классификатор*.

SFC-схема позволяет визуализировать процесс в виде последовательности шагов и увидеть возможные пути выполнения алгоритма. Это является очень удобным и помогает при проектировании, разработке и отладке.



## 2. Основы SFC

### 2.1. Шаги и переходы

SFC позволяет представить программу в виде последовательности шагов, связанных переходами. Каждый шаг характеризуется выполняемыми действиями, а каждый переход – условием.

Шаг описывает одну из стадий технологического процесса. В каждый момент времени шаг является либо активным, либо неактивным. Ниже приведено графическое и текстовое представление шага:

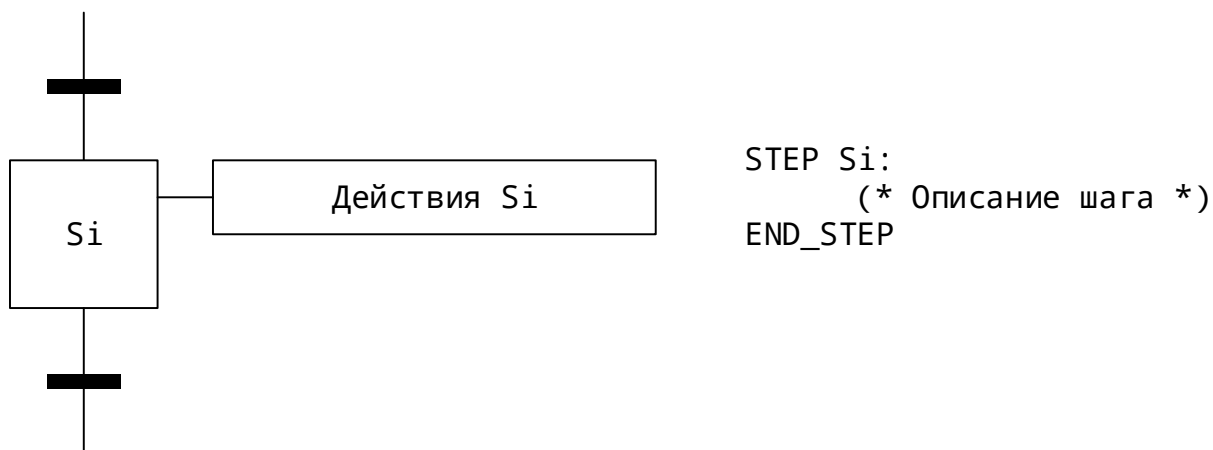


Рис. 1. Графическое и текстовое представление SFC-шага

В реальных проектах шаги имеют осмысленные названия (например, Mixing).

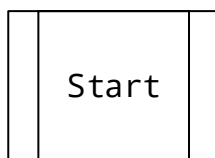
Шаг обладает неявными переменными, скрытыми за пространством имен, совпадающим с названием шага:

- `<имя_шага>.X` – флаг активности шага (TRUE – активен, FALSE – неактивен);
- `<имя_шага>.T` – время, прошедшее с момента активации шага (если шаг неактивен, то переменная сохраняет свое последнее значение. При активации шага значение сбрасывается до T#0s<sup>1</sup>).

Для шага с рис. 1 данные переменные, соответственно, будут иметь имена **Si.X** и **Si.T**. Имя шага и его переменных являются локальными для POU, в котором объявлен шаг.

Начальное состояние POU определяется значениями его переменных, а также набором начальных шагов (т.е. шагов, которые активны при запуске POU). Каждая SFC-схема имеет только один начальный шаг. В графическом представлении он выделяется двойным контуром, в текстовом – ключевым словом **INITIAL\_STEP**. При инициализации приложения флаги активности обычных шагов имеют значение FALSE, а флаги начальных шагов – TRUE.

<sup>1</sup> В CODESYS V3.5 переменная сбрасывается в T#0s при деактивации шага (прим. пер.)



Переход описывается условием, при выполнении которого активным становится следующий шаг(-и), а текущий шаг прекращает активность. Переходы графически соединяют нижнюю часть предыдущего шага и верхнюю часть шага, следующего за ним; другие варианты соединения шагов невозможны. Условие перехода представляет собой логическое выражение. Если необходимо, чтобы условие всегда было истинным, то можно в качестве него использовать литерал TRUE.

В графическом представлении переходы изображаются вертикальными линиями, соединяющими блоки шагов. Условие перехода может быть описано одним из следующих способов:

- логическим выражением на языке ST;
- цепью на языке LD или FBD;
- конструкцией **TRANSITION...END\_TRANSITION**, которая включает в себя:
  - ключевое слово **TRANSITION FROM**, дополненное именем шага (или списка шагов), с которого производится переход;
  - ключевое слово **TO**, дополненное именем шага (или списка шагов), на который производится переход;
  - логическое выражение на языке ST или IL;
  - ключевое слово **END\_TRANSITION**.
- названием перехода, которое ссылается на именованную конструкцию **TRANSITION...END\_TRANSITION**, содержащую логическое выражение на языке LD/FBD/IL/ST.

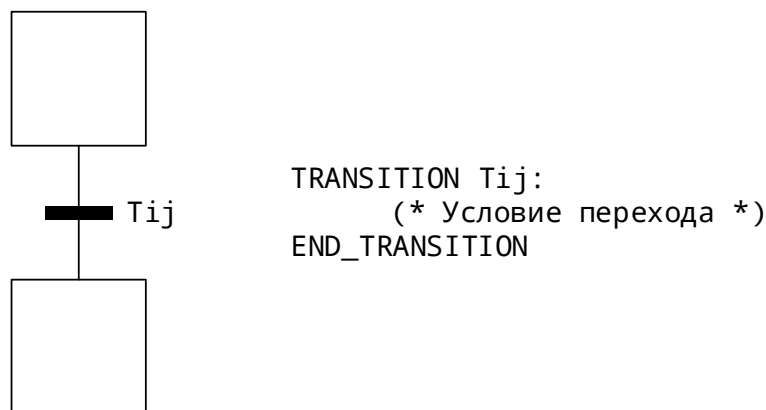


Рис. 2. Графическое и текстовое представление перехода

Имя перехода является локальным для ROU, в котором объявлен шаг. Во время проверки условия перехода не может возникнуть никаких побочных эффектов (например, присвоения значений переменным).

## 2.2. Действия

С каждым шагом может быть связано одно или несколько действий. Если шаг не содержит действий, то программа просто ожидает выполнения условия перехода к следующему шагу. Действие имеет имя и тело (реализацию), которое может быть закодировано на любом из языков стандарта МЭК 61131-3 (в т.ч. SFC<sup>2</sup>). Графически действие представляется блоком, расположенным справа от шага и связанным с ним горизонтальной линией. Имя действия является локальным для ROU, в котором объявлен шаг. Объявление действия сопровождается неявным созданием экземпляра ФБ [ACTION CONTROL](#), который позволяет управлять процессом его выполнения.

## 2.3. Классификаторы действий

Каждое действие имеет классификатор, который определяет способ влияния активного шага на выполнение действия (например, действие с классификатором N будет выполняться, пока активен связанный с ним шаг). Классификаторы L, D, SD, DS и SL требуют дополнительного значения типа TIME для определения продолжительности действия. Более подробная информация о классификаторах приведена в [п. 3.5](#).

Классификатор	Описание действия	Условие выполнения действия
None	Пустой классификатор	Действие выполняется, пока активен данный шаг.
N	Non-stored (несохраняемое)	Действие выполняется, пока активен данный шаг.
P	Pulse (импульс)	Действие однократно выполняется при активации и деактивации шага.
S	Stored (сохраняемое)	Действие выполняется до тех пор, пока не будет сброшено R-действием (даже если данный шаг уже не является активным).
R	Reset (сброс)	Сброс сохраняемого действия (S, SD, DS, SL).
L	Time Limited (ограничение по времени)	Действие активируется вместе с шагом и остается активным заданное время, но не дольше, чем шаг.
D	Delayed (отложенное)	Действие активируется через заданное время после активации шага и остается активным, пока активен шаг. Если шаг окажется активным меньше заданного времени, то действие не будет активировано.
SD	Stored and Delayed (сохраняемое отложенное)	Действие активируется через заданное время после активации шага, даже если шаг уже неактивен. Но если в процессе отсчета задержки активации выполнить сброс действия (в другом шаге с классификатором R), то активации не произойдет. Активированное действие остается активным до сброса.
DS	Delayed and Stored (отложенное сохраняемое)	Действие активируется через заданное время после активации шага и остается активным до сброса. Если шаг активен меньше заданного времени, то действие не будет активировано. При выполнении сброса в процессе отсчета времени (в другом шаге с классификатором R) действие не будет активировано.
SL	Stored and time Limited (сохраняемое и ограниченное по времени)	Действие активируется вместе с шагом и остается активным заданное время вне зависимости от активности шага. Действие можно деактивировать досрочно из другого шага с классификатором R.

<sup>2</sup> Тем не менее, на самом низком уровне вложенности для кодирования действий должен использоваться язык программирования, отличный от SFC (прим. пер.)

## 2.4. Обработка SFC-схемы

Обработка SFC-схемы выполняется с начального шага, который становится активным сразу при вызове POU. Далее активность шагов распространяется через прямые связи при выполнении соответствующих условий перехода. Переход становится активным, если активны все предшествующие ему шаги (т.е. шаги, связанные с ним вертикальными линиями). Если условие перехода выполняется, то следующий за ним шаг становится активным.

После выполнения перехода предшествующие ему шаги становятся неактивными. Этот процесс называется очисткой. Время очистки достаточно мало, но не может считаться нулевым. На практике это время зависит от конкретного ПЛК. По этой же причине время активности любого из шагов не может считаться нулевым. Если несколько переходов могут быть очищены одновременно, то это должно произойти в пределах одного цикла ПЛК. Проверка условия перехода должна выполняться только после того, как эффект активации предшествующих шагов распространился по всей SFC-схеме.

## 2.5. Расходимость и сходимость

Расходимостью называется связь SFC-символа (шага или перехода) с двумя или более символами другого общего типа (переходами или шагами). Сходимостью называется связь нескольких SFC-символов общего типа с символом другого типа. Расходимость и сходимость могут быть альтернативными или параллельными.

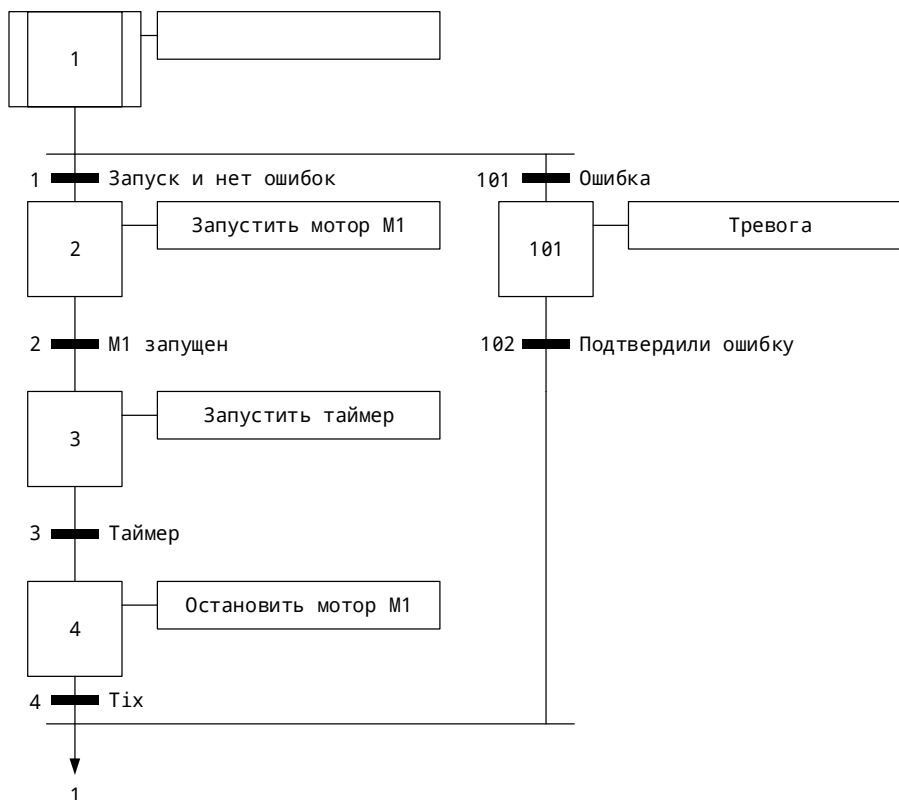


Рис. 3. SFC-схема с альтернативной расходимостью и сходимостью

Альтернативная расходимость представляет собой связь одного шага с несколькими переходами. На рис. 3 выполняется только один из переходов (1 или 101), так как их условия являются взаимоисключающими (см. также [п. 4.3](#)). Тем не менее, условия переходов в точке расхождения не обязательно исключают друг друга, поэтому выбор исполняемого перехода (который может быть только один) должен контролироваться пользователем (например, с помощью добавления переходов в порядке снижения их приоритета) или настройками среды программирования. Сходимость для альтернативной расходимости представляет собой связь нескольких переходов с одним шагом. Такая сходимость обычно используется для SFC-ветвей, полученных при одиночной расходимости.

Альтернативная сходимость и расходимость изображаются на SFC-схеме с помощью горизонтальной линии (как на рис. 3). Частным случаем альтернативной расходимости является *пустая ветвь* – так называется SFC-ветвь, не содержащая шагов. Другим частным случаем является *ветвь с циклом*, которая содержит переходы на предыдущие (расположенные выше по схеме) шаги.

Параллельная расходимость представляет собой связь одного перехода с несколькими шагами. После выполнения условия перехода первые шаги всех параллельных ветвей становятся активными. Такой подход используется для описания операций процесса управления, которые выполняются одновременно. Сходимость для параллельной расходимости представляет собой связь нескольких шагов с одним переходом. Такая сходимость обычно используется для SFC-ветвей, полученных при двойной расходимости. Параллельная сходимость выполняется в том случае, если все предшествующие ей шаги являются активными, а условие перехода – истинным. После схождения ветвей все предшествующие шаги деактивируются, и активным становится шаг, расположенный после перехода.

Параллельная сходимость и расходимость изображаются на SFC-схеме с помощью двойной горизонтальной линии (как на рис. 4). Пример правильной организации параллельной расходимости рассмотрен в [п. 4.3](#) вместе с анализом типичных ошибок, возникающих при ее использовании.

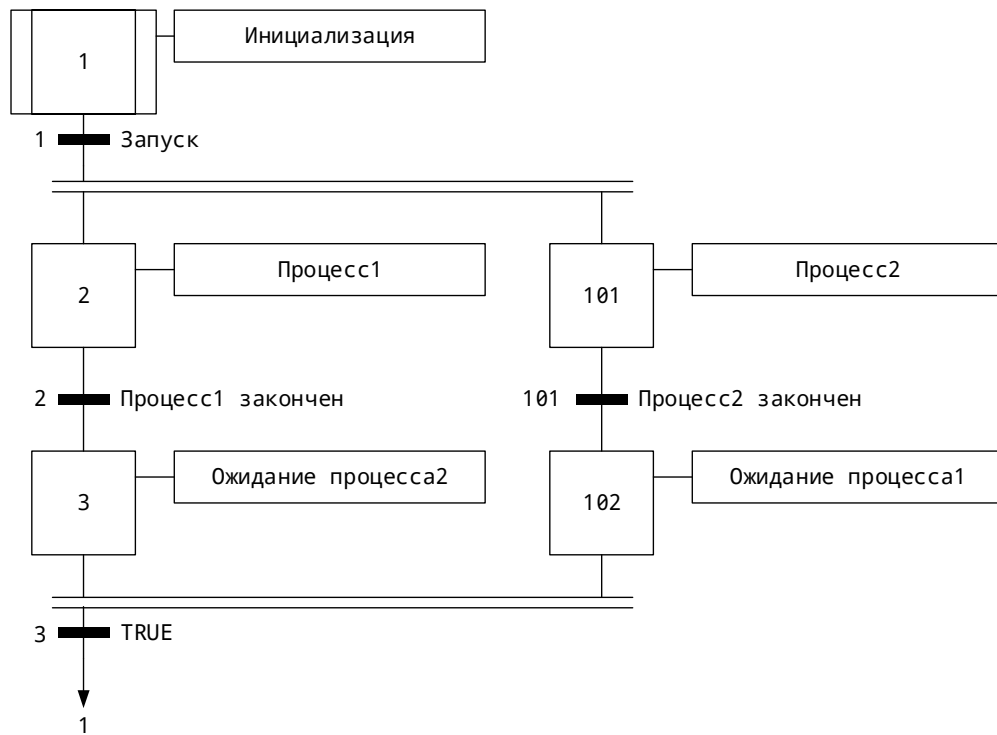
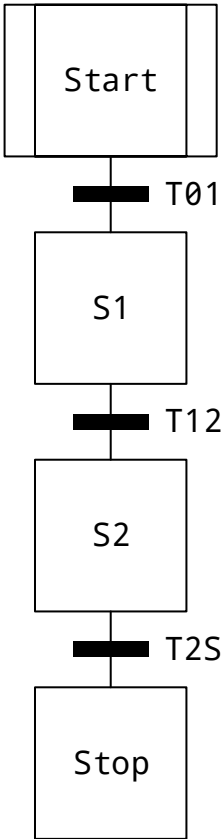


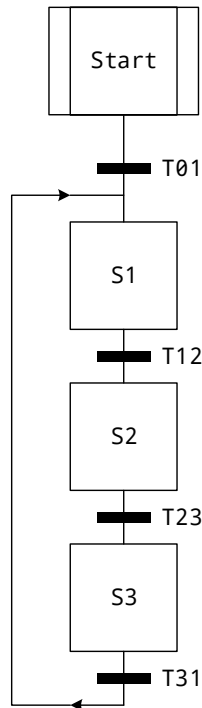
Рис. 4. SFC-схема с параллельной расхожимостью и сходимостью

### 3. Структурные свойства SFC

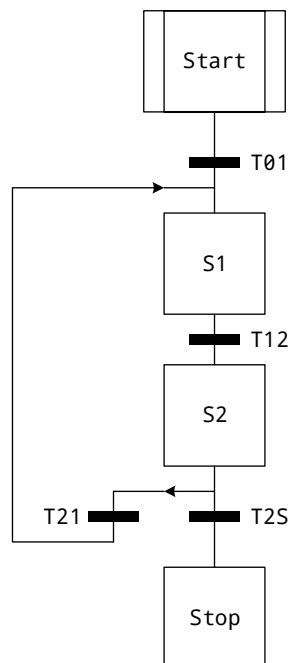
#### 3.1. Структура процесса управления

Структура простейшего алгоритма управления представляет собой набор последовательно выполняемых шагов (как на рисунке ниже). Начальный шаг, выделенный двойным контуром, автоматически начинается выполняться при первом вызове ПОУ. Затем, по мере выполнения условий перехода, последовательно активируются остальные шаги (от верхнего к нижнему).

Последовательность шагов	Простейшая форма SFC-схемы.
 <p>The diagram illustrates a sequence of four steps: Start, S1, S2, and Stop. The 'Start' step is enclosed in a double-line border, indicating it is the initial step. It is connected to step S1 by transition T01. S1 is connected to S2 by transition T12. S2 is connected to the final step, Stop, by transition T2S.</p>	<ul style="list-style-type: none"><li>• При запуске приложения начальный шаг Start является активным. При выполнении условия перехода T01 шаг Start деактивируется, и активным становится шаг S1.</li><li>• При выполнении условия перехода T12 шаг S1 деактивируется, и активным становится шаг S2.</li><li>• При выполнении условия перехода T2S шаг S2 деактивируется, и активным становится шаг Stop. Поскольку шаг Stop является конечным элементом SFC-схемы, то он остается активным до перезапуска приложения.</li></ul>

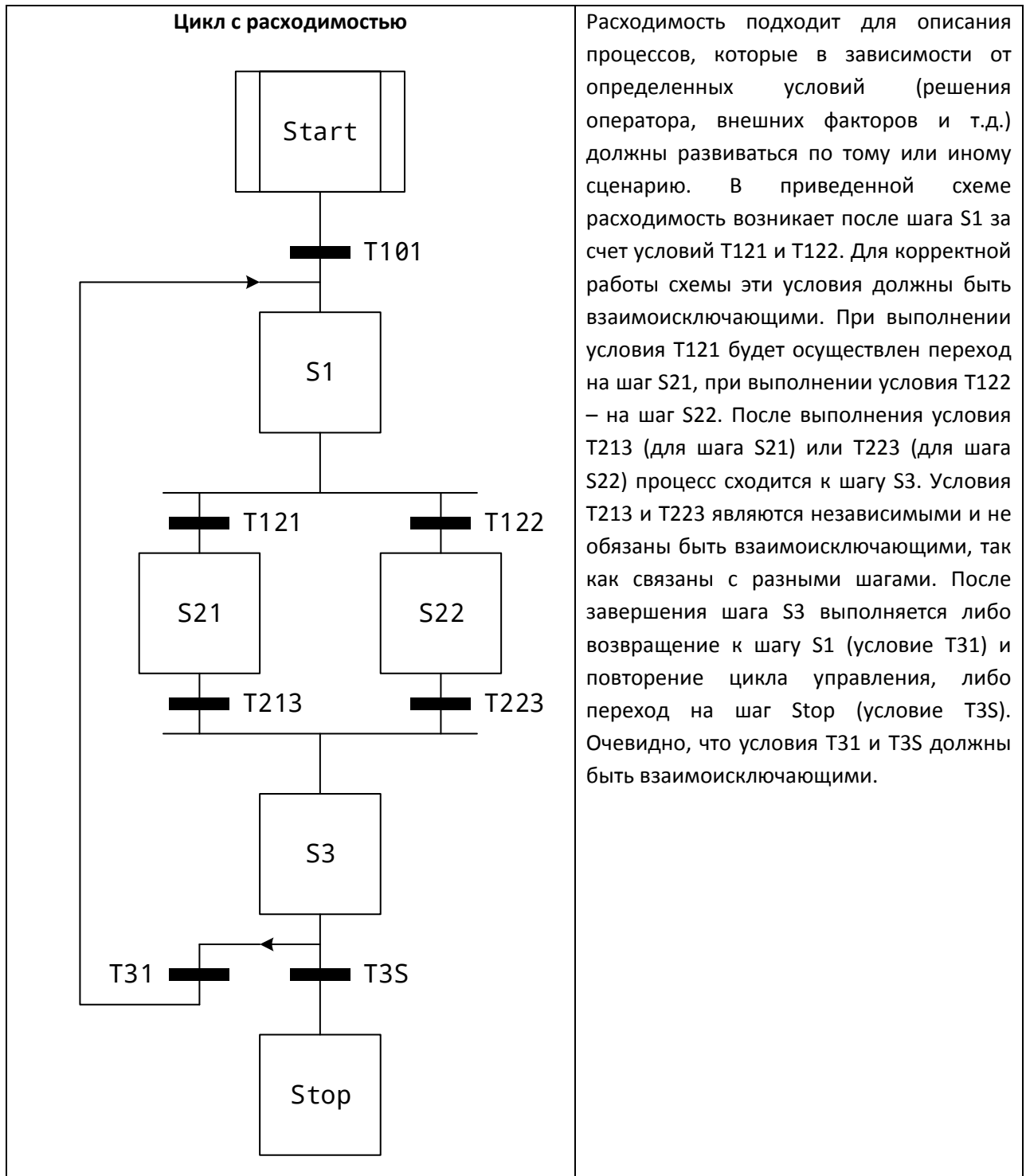
**Цикл без условия выхода**

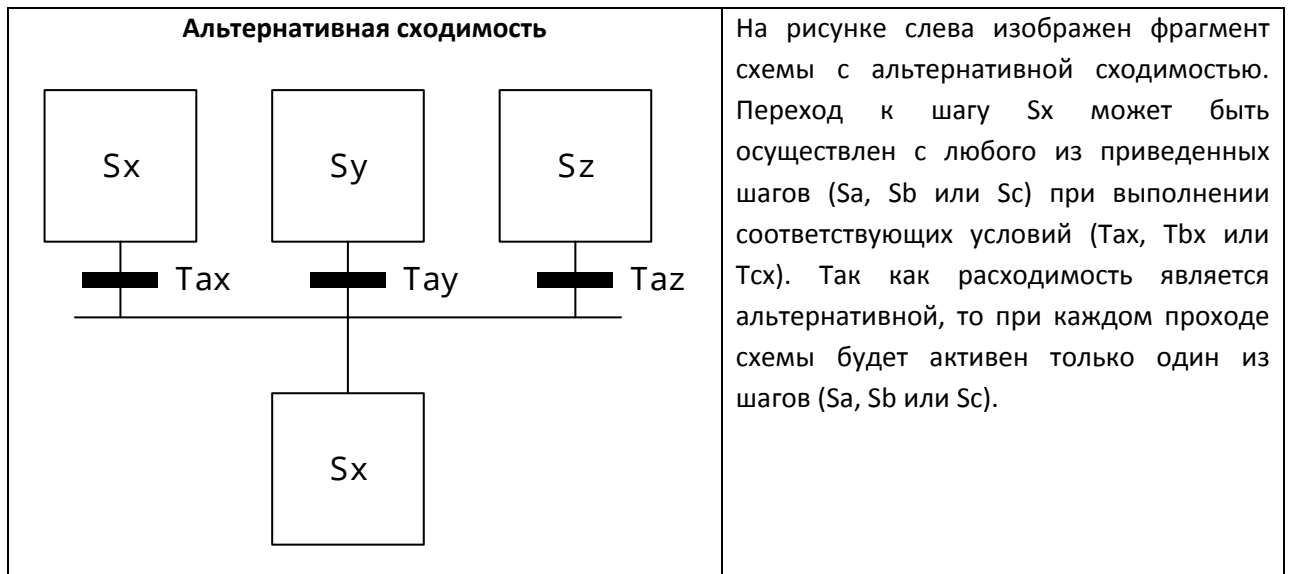
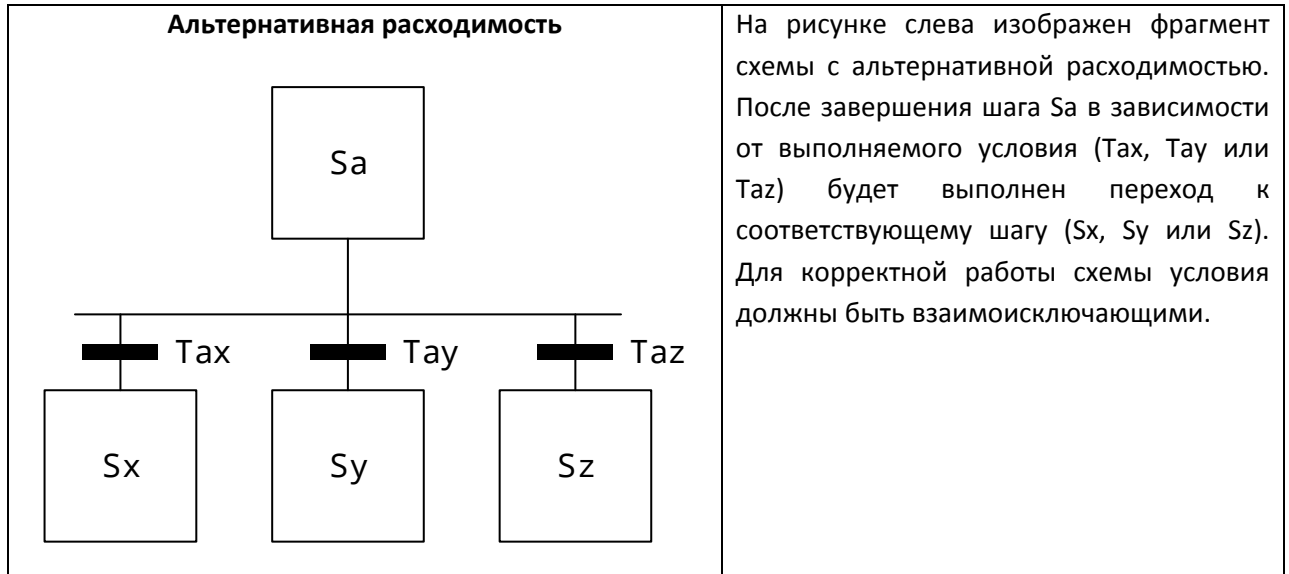
В значительном числе случаев требуется не однократное, а циклическое выполнение шагов. Как и в предыдущем варианте, шаги S1/S2/S3 последовательно активируются, но после выполнения условия T31 происходит повторная активация шага S1, и процесс выполнения начинается заново. В рамках приведенной схемы нет условия, которое бы прекратило выполнение этого цикла. Подразумевается, что данная схема является содержимым шага SFC-схемы верхнего уровня. Таким образом, при деактивации этого высокоуровневого шага выполнение данного цикла также прекратится.

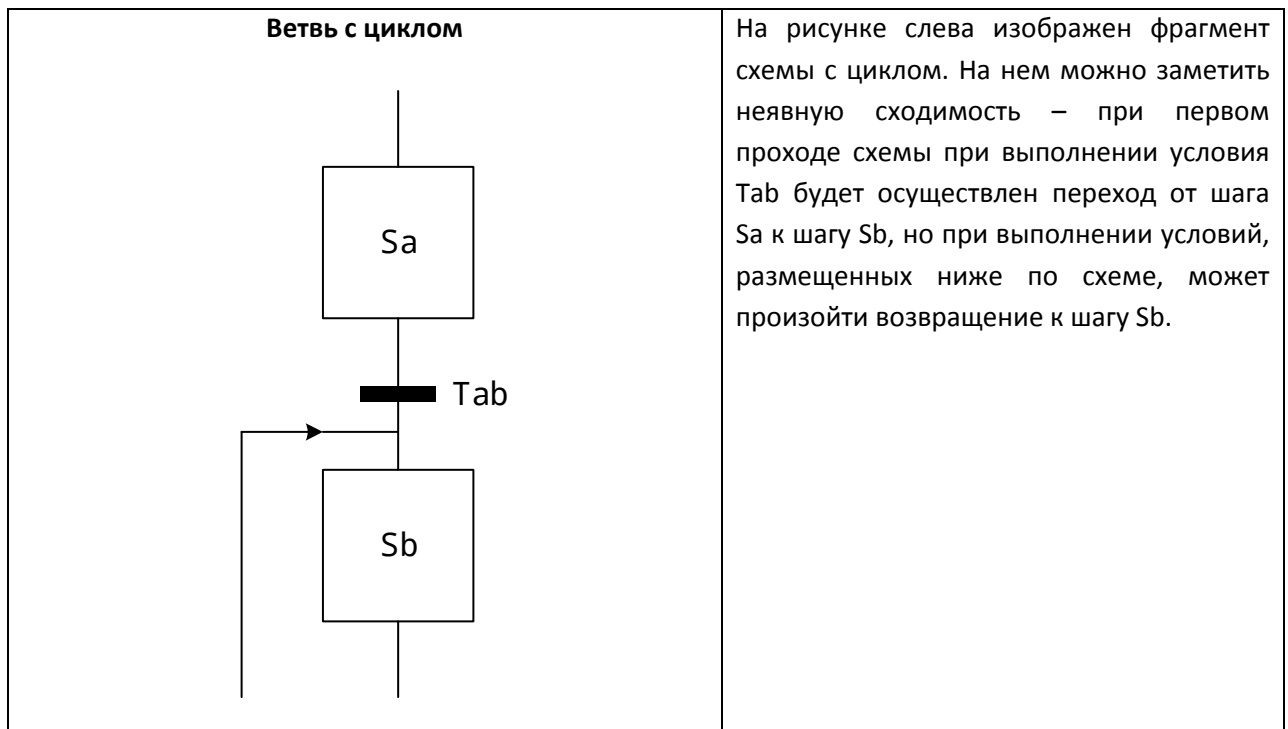
**Цикл с условием выхода**

В приведенной схеме цикл имеет условие выхода. Пока выполняется условие T21 шаги S1 и S2 последовательно выполняются в цикле. После выполнения условия T2S будет осуществлен переход на шаг Stop, который останется активным до перезапуска приложения. Для корректной работы схемы условия T21 и T2S должны быть взаимоисключающими. Схему хорошо иллюстрирует следующий практический пример: пусть имеется конвейер, обрабатывающий заготовки. После запуска в работу (Start) конвейер дожидается новой заготовки (S1), обрабатывает ее (S2), после чего дожидается следующей заготовки (S1) и т.д. В определенный момент времени конвейер необходимо остановить (Stop). В этом случае следует выполнить условие T2S, которое приведет к прекращению техпроцесса.







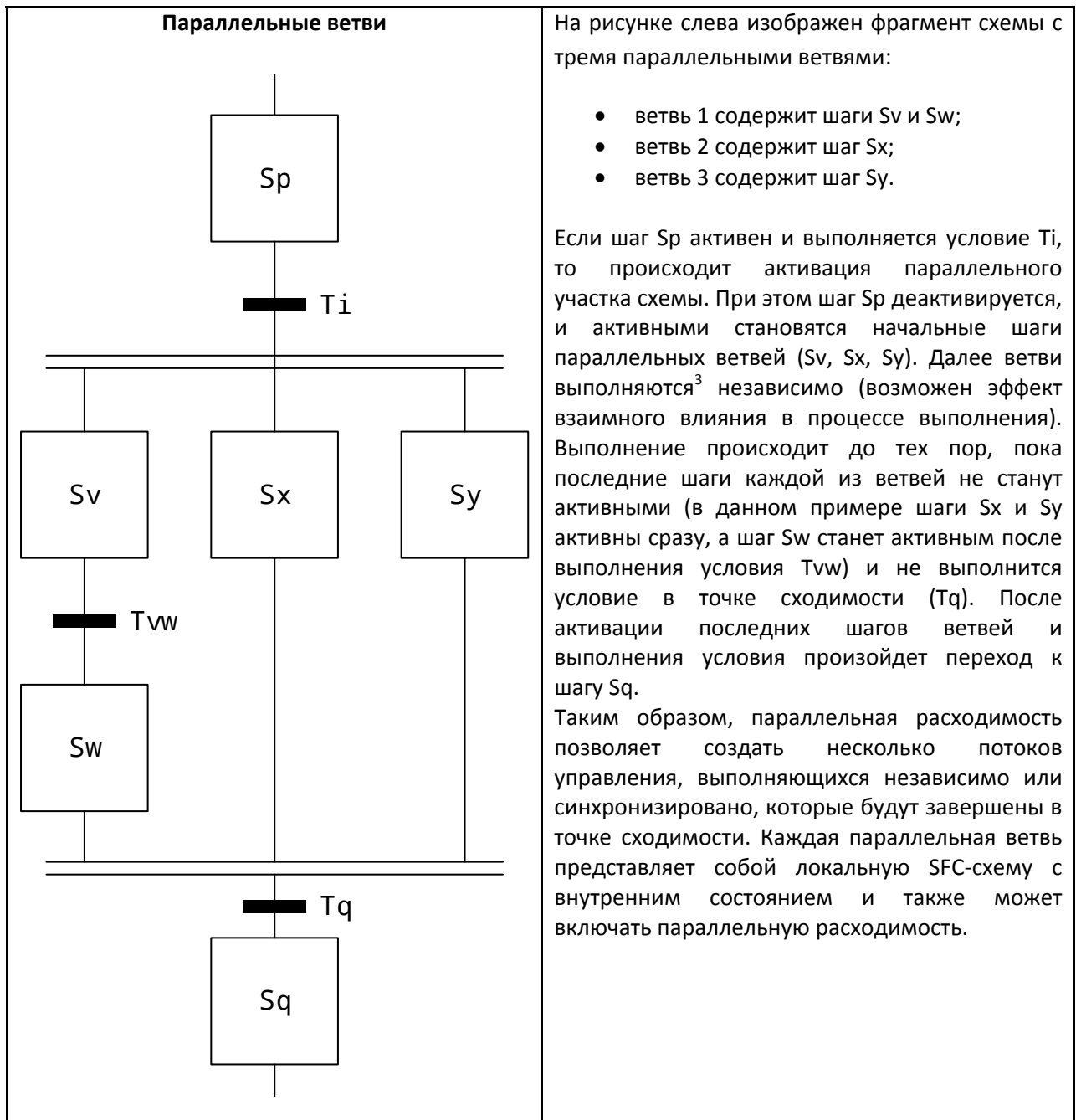


### 3.2. Параллельные процессы

Язык SFC позволяет описывать процессы, протекающие параллельно. Это означает, что в некий момент времени будут активны два или более шагов SFC-схемы. Участки схем с параллельными ветвями начинаются с расходимости и завершаются сходимостью. Стоит отметить два ключевых момента:

- Активация параллельных ветвей происходит в тот момент, когда предыдущий шаг является активным, и выполняется условие перехода, расположенное после него. При этом одновременно становятся активными начальные шаги всех параллельных ветвей и происходит деактивация предыдущего шага;
- Параллельные ветви завершаются одновременно в точке сходимости, когда последние шаги каждой из ветвей являются активным и выполняется условие перехода к следующему шагу. При этом последние шаги ветвей деактивируются, и активным становится шаг, расположенный за ними.

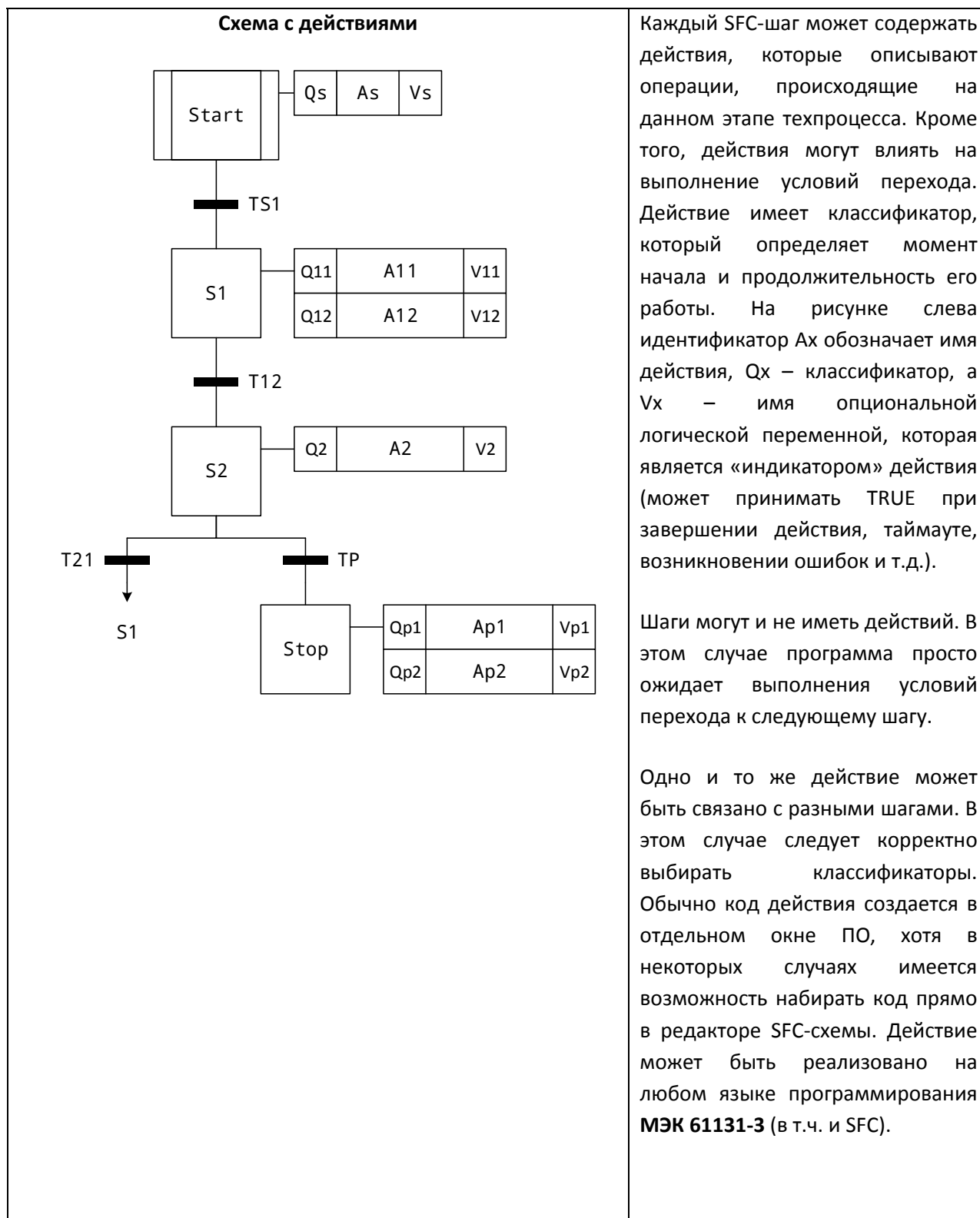
## 3.3. Параллельность в SFC-схеме



<sup>3</sup> В среде CODESYS 3.5 параллельные ветви выполняются в пределах одного цикла ПЛК, слева направо (прим. пер.)

### 3.4. Действия

#### 3.4.1. Вступление



### 3.5. Классификаторы действий

#### 3.5.1. Вступление

Каждое действие имеет классификатор, который определяет, в какой момент действие начнет выполняться и какова будет продолжительность его выполнения. Классификатор устанавливает зависимость между активностью шага и выполнением связанных с этим шагом действий. По этой причине каждый шаг имеет две неявные переменные, скрытые за пространством имен, совпадающим с названием шага:

- <имя\_шага>.X – флаг активности шага (TRUE – активен, FALSE – неактивен);
- <имя\_шага>.T – время, прошедшее с момента активации шага (если шаг неактивен, то переменная сохраняет свое последнее значение. При активации шага значение сбрасывается до T#0s<sup>4</sup>).

Обе этих переменных могут использоваться в коде действий и условий всех шагов схемы (а не только того, к которому они относятся) – это позволяет синхронизировать выполнение операций.

Стандарт **МЭК 61131-3** определяет следующие классификаторы действий (табл. 59):

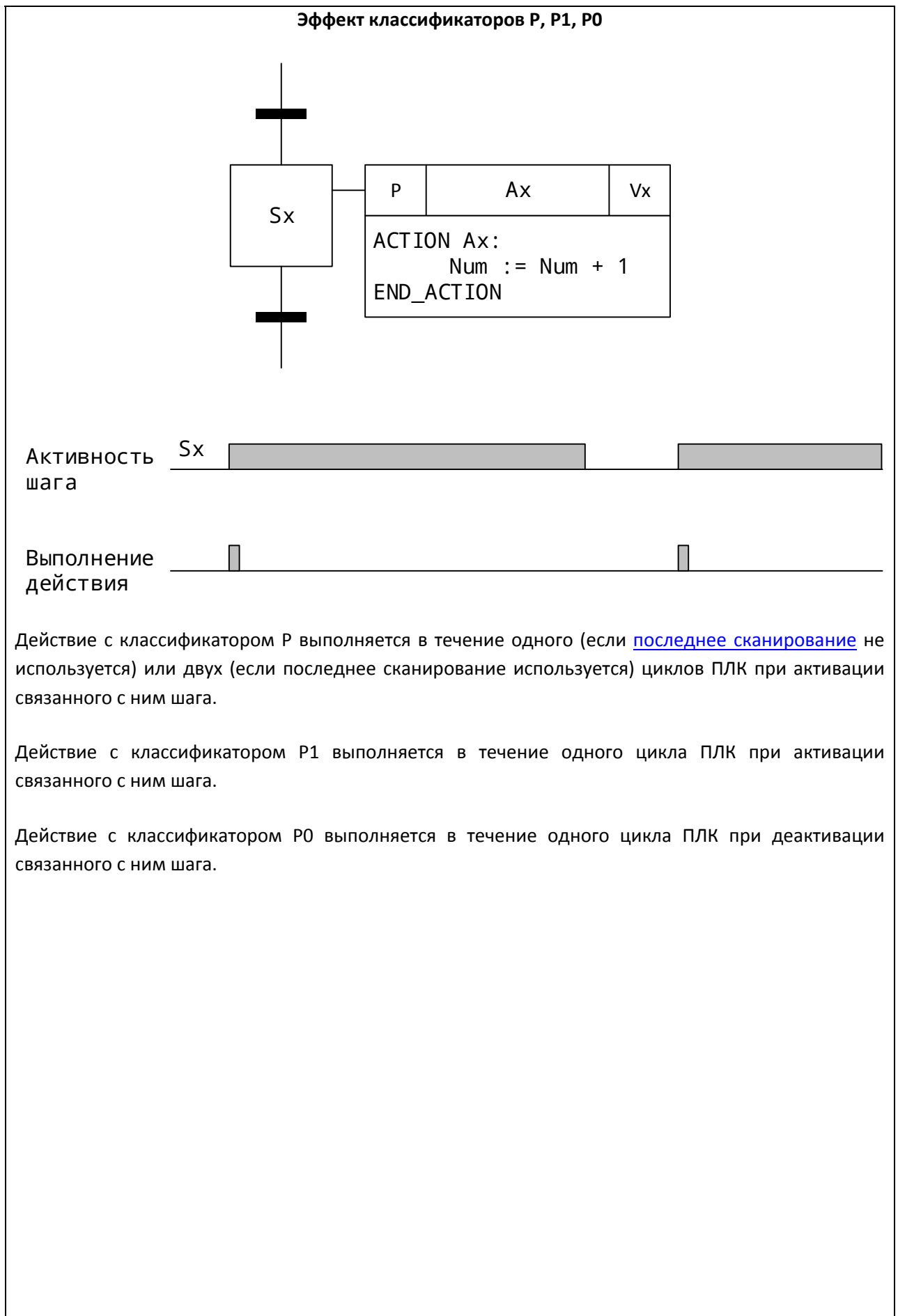
- None (пустой классификатор)
- N (несохраняемое действие)
- R (сброс действия)
- S (сохраняемое действие)
- L (действие с ограничением по времени)
- D (отложенное действие)
- P (выполнение действия по импульсу)
- SD (сохраняемое отложенное действие)
- DS (отложенное сохраняемое действие)
- SL (сохраняемое действие с ограничением по времени)
- P1 (выполнение действия по переднему фронту)
- P0 (выполнение действия по заднему фронту)

Если условие перехода становится истинным, то существует два варианта обработки действий – с последним сканированием и без последнего сканирования. Более подробная информация приведена в [п. 3.6.2](#).

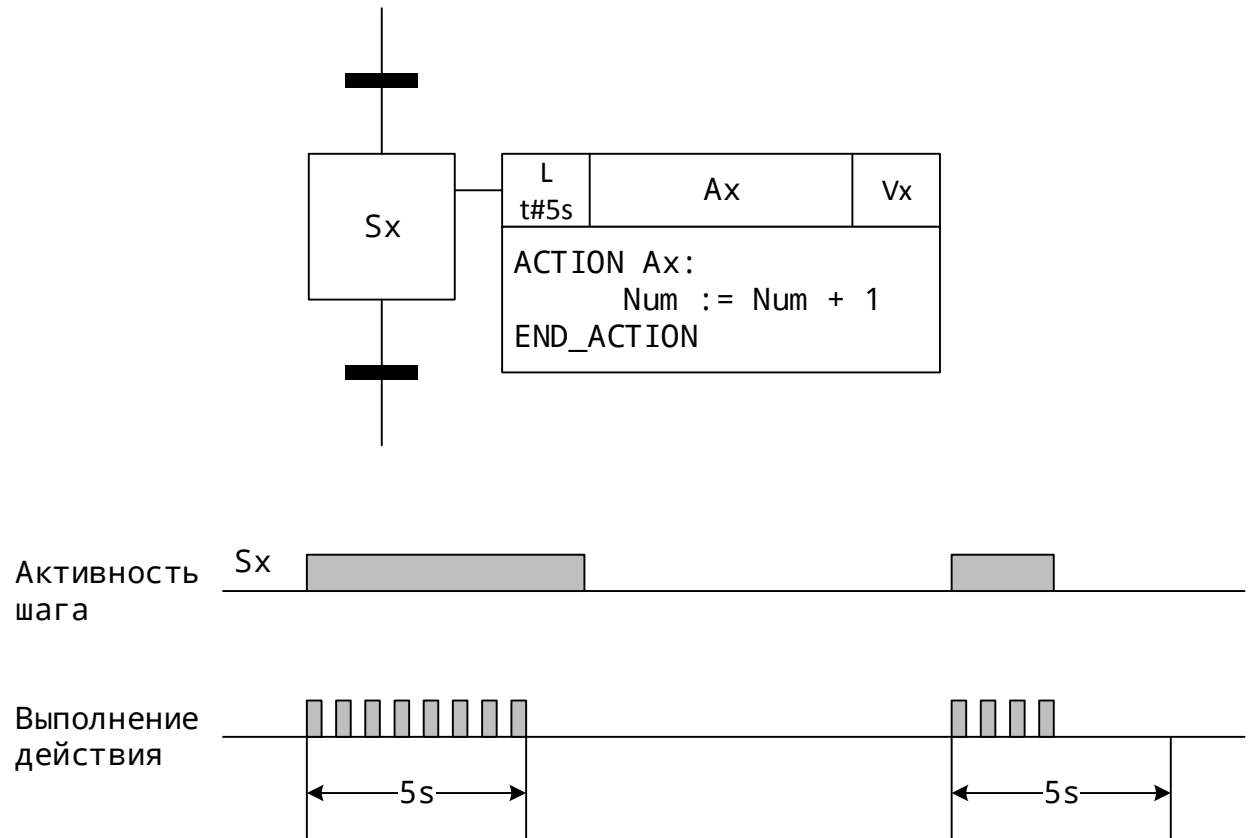
Описание классификаторов действий приведено ниже.

---

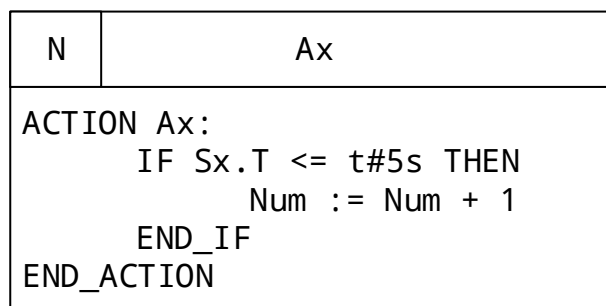
<sup>4</sup> В CODESYS V3.5 переменная сбрасывается в T#0s при деактивации шага (прим. пер.)



## Эффект классификатора L



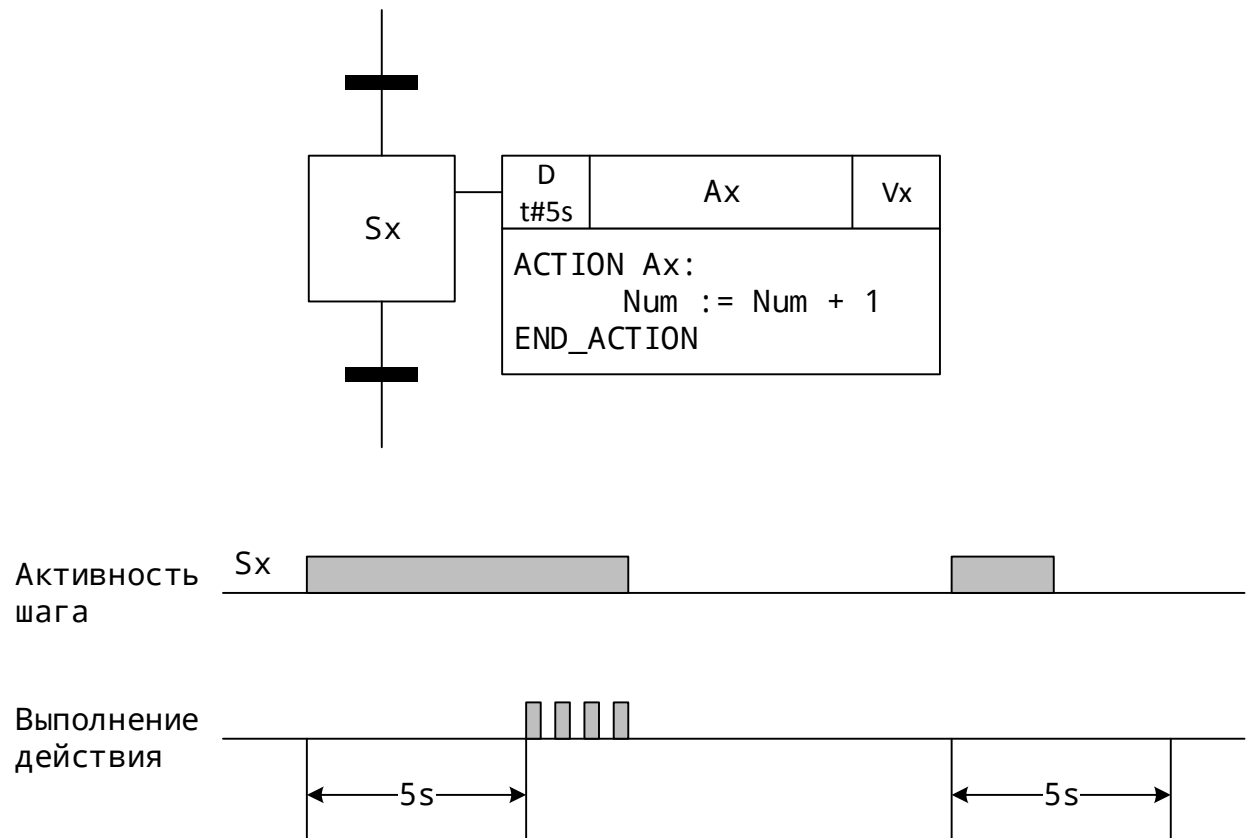
Действие с классификатором L начинает выполняться при активации связанного с ним шага и выполняется в каждом цикле ПЛК в течение заданного времени (на рис. выше это время ограничено 5 секундами). Если шаг деактивируется до истечения заданного времени, то выполнение действия прекращается (см. вторую активацию шага на рис. выше). Выполнение действия с классификатором L эквивалентно действию с классификатором N, в коде которого производится контроль времени активности шага через неявную переменную:



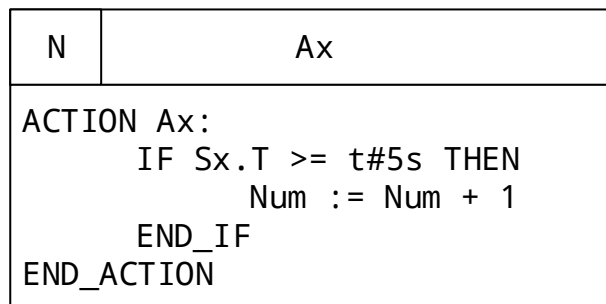
**Примечание:** здесь и далее в операциях сравнения могут использоваться операторы «больше/меньше» или «больше или равно/меньше или равно». Выбор конкретного оператора зависит от реализации обработки классификатора в той или иной среде программирования.



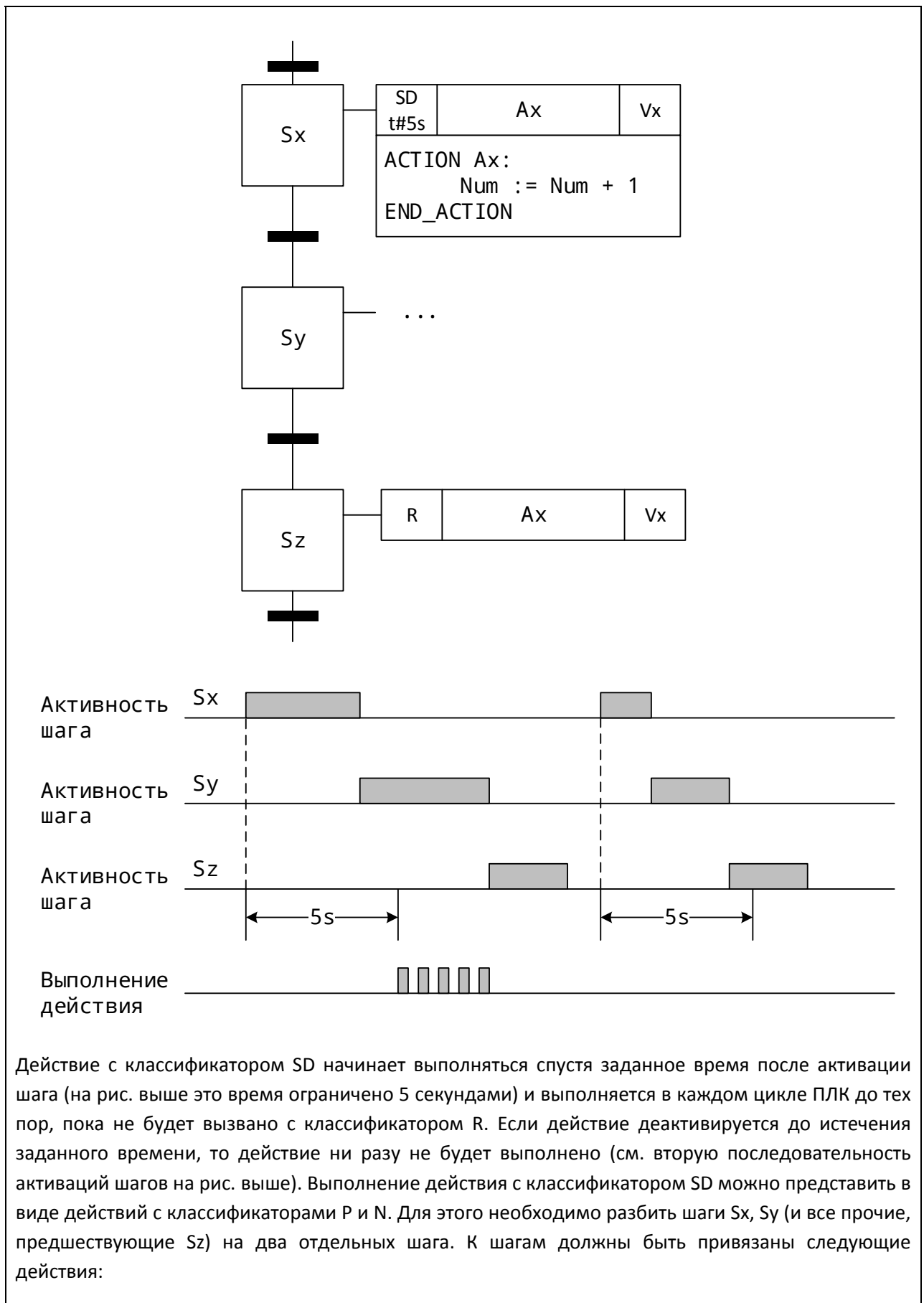
## Эффект классификатора D

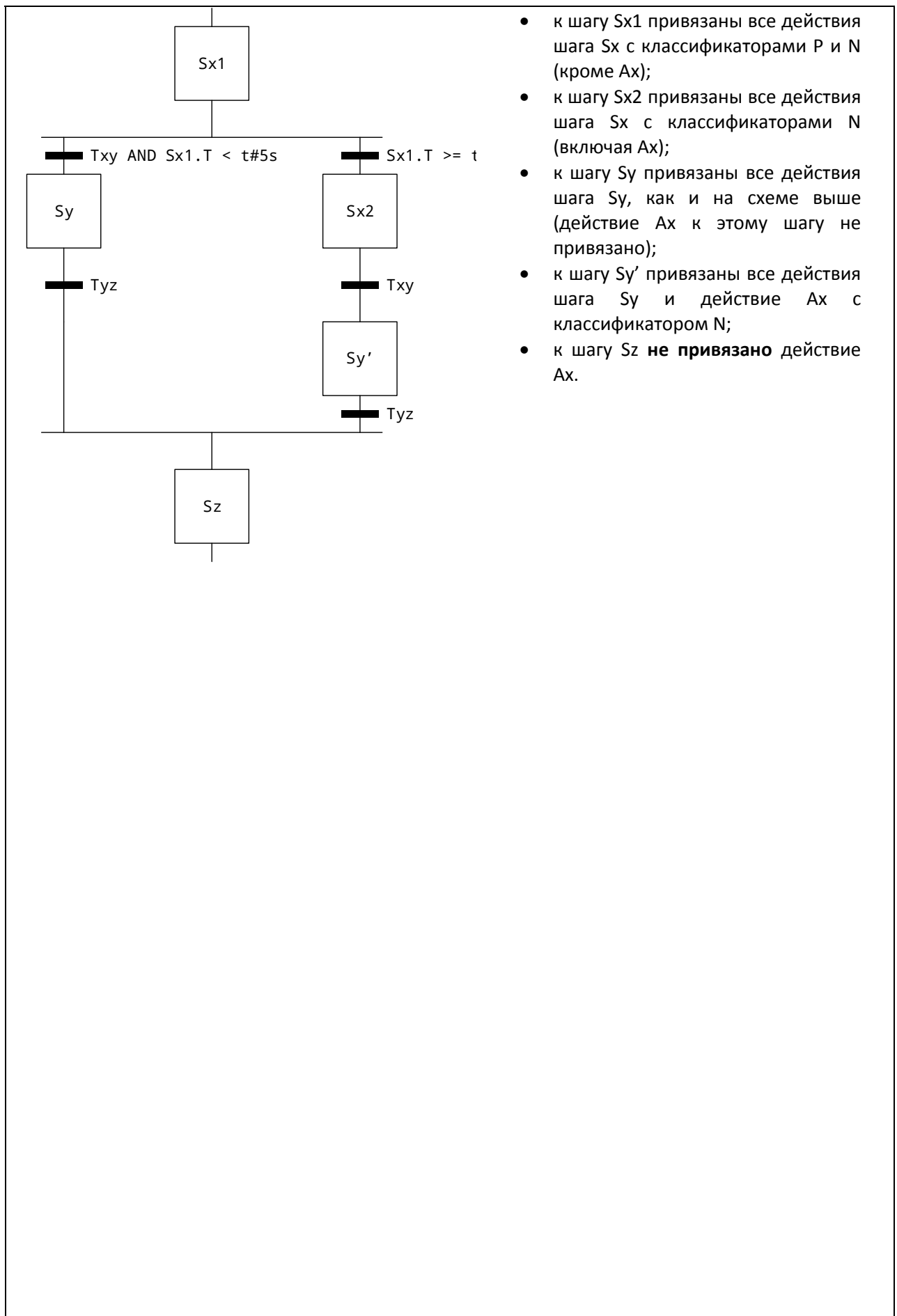


Действие с классификатором D начинает выполняться спустя заданное время после активации шага (на рис. выше это время ограничено 5 секундами) и выполняется в каждом цикле до тех пор, пока шаг не будет деактивирован. Если шаг деактивируется до истечения заданного времени, то действие ни разу не будет вызвано (см. вторую активацию шага на рис. выше). Выполнение действия с классификатором D эквивалентно действию с классификатором N, в коде которого производится контроль времени активности шага через неявную переменную:



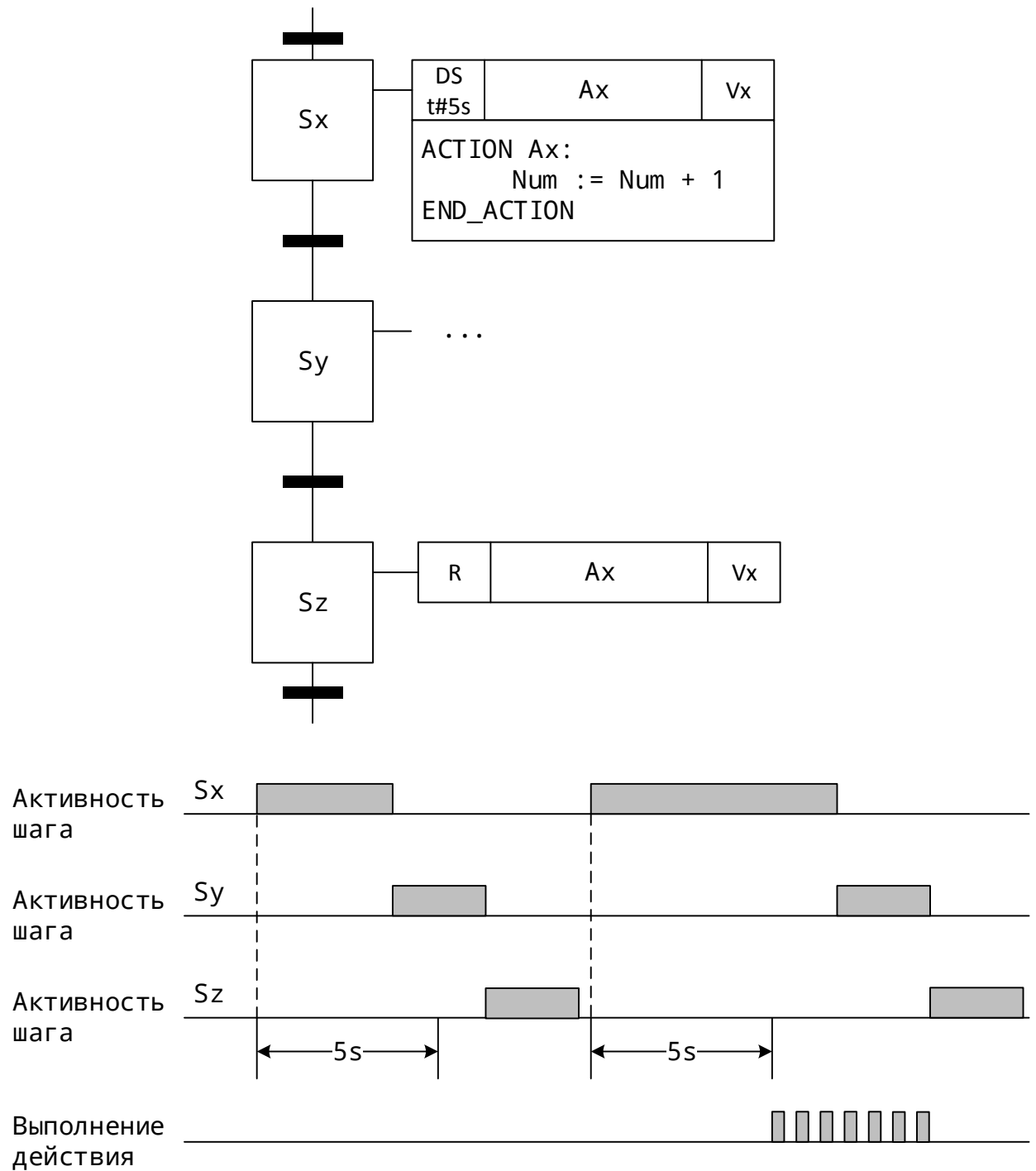
## Эффект классификатора SD





- к шагу Sx1 привязаны все действия шага Sx с классификаторами P и N (кроме Ax);
- к шагу Sx2 привязаны все действия шага Sx с классификаторами N (включая Ax);
- к шагу Sy привязаны все действия шага Sy, как и на схеме выше (действие Ax к этому шагу не привязано);
- к шагу Sy' привязаны все действия шага Sy и действие Ax с классификатором N;
- к шагу Sz **не привязано** действие Ax.

## Эффект классификатора DS



Действие с классификатором DS начинает выполняться спустя заданное время после активации шага (если шаг к этому моменту остается активен) и выполняется в каждом цикле ПЛК до тех пор, пока не будет вызвано с классификатором R. Если действие вызывается с классификатором R до

истечения заданного времени, то его код ни разу не будет выполнен (см. первую последовательность активаций шагов на рис. выше). Выполнение действия с классификатором DS можно представить в виде действий с классификаторами P и N. В данном случае использовать неявную переменную Sx.T нельзя, так как выполнение действия не связано с активностью шага. Вместо этого к шагу Sx необходимо привязать действие, в котором будет вызываться экземпляр таймера TON:

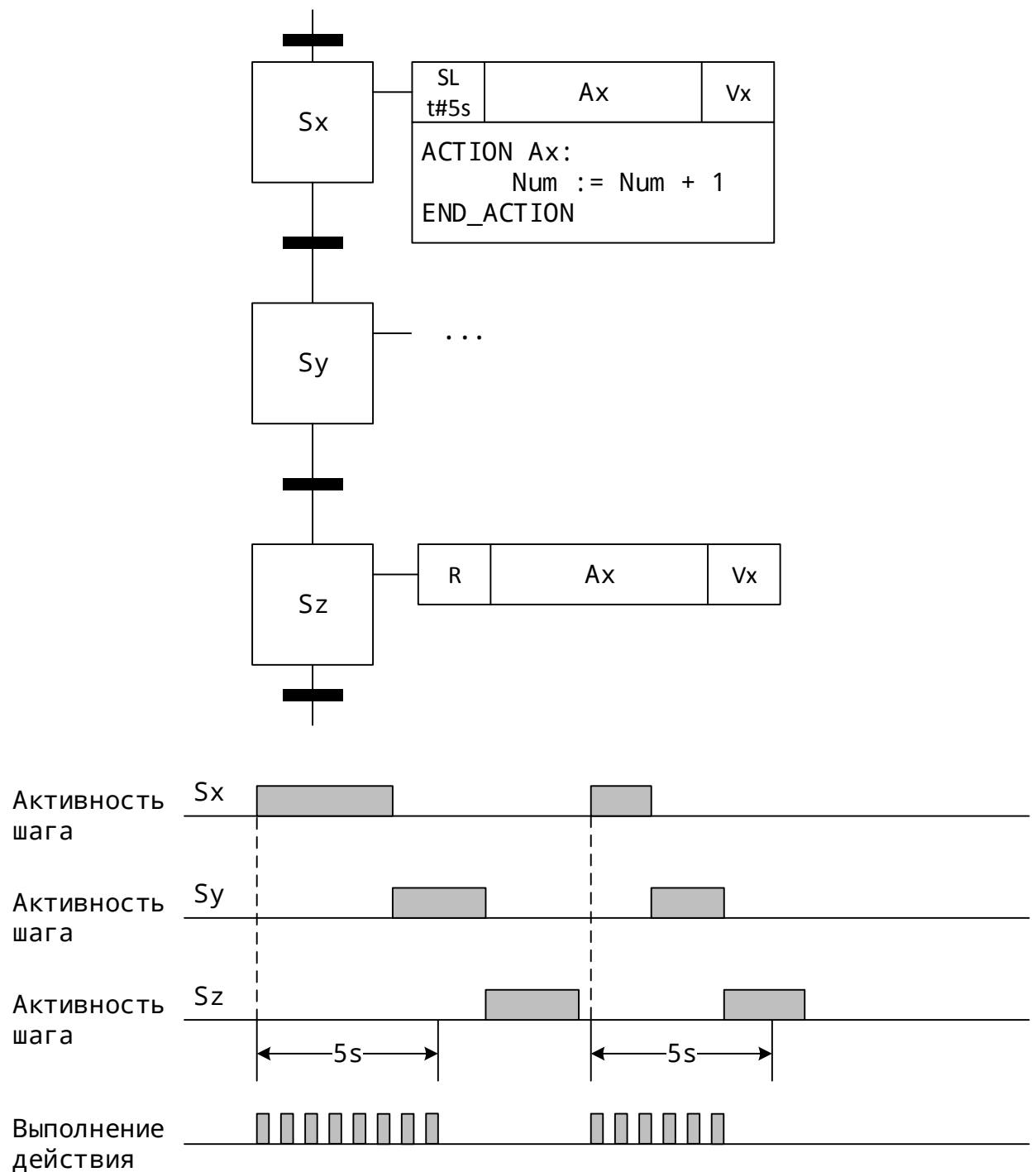
P	Ax1
<pre> ACTION Ax1:     TONx (IN := TRUE, PT := t#5s) END_ACTION </pre>	

К каждому следующему шагу, где должен выполняться нужный код, необходимо привязать действие Ax2, в котором будет осуществляться проверка таймера<sup>5</sup>:

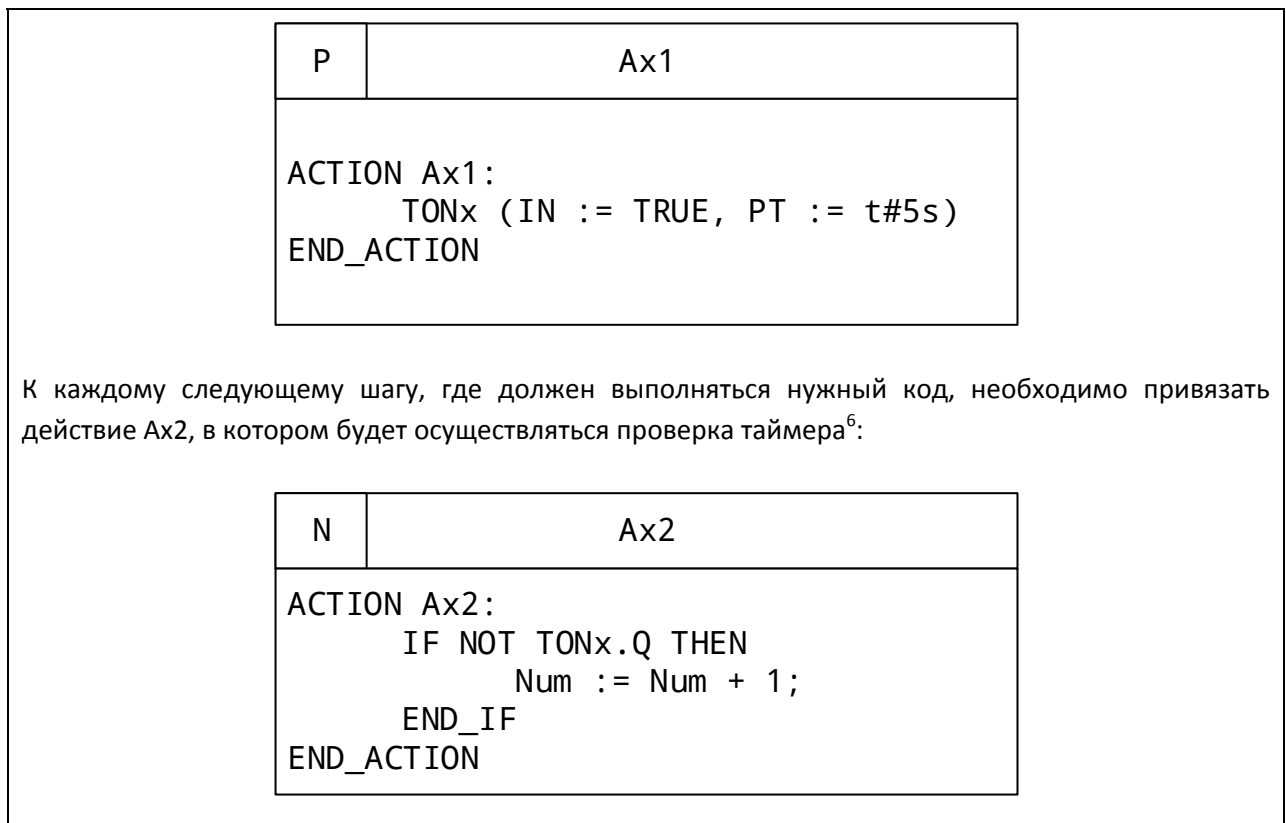
N	Ax2
<pre> ACTION Ax2:     IF TONx.Q THEN         Num := Num + 1;     END_IF END_ACTION </pre>	

<sup>5</sup> Перед запуском таймер должен быть сброшен с помощью вызова с IN:=FALSE (прим. пер.)

## Эффект классификатора SL



Действие с классификатором SL начинает выполняться при активации связанного с ним шага и выполняется в каждом цикле ПЛК в течение заданного времени (даже если шаг уже не является активным) или до его вызова с классификатором R (см. вторую последовательность активаций шагов на рис. выше). Выполнение действия с классификатором SL можно представить в виде действий с классификаторами P и N. Как и в случае с классификатором DS, использовать неявную переменную  $S_x.T$  нельзя, так как выполнение действия не связано с активностью шага. Вместо этого к шагу  $S_x$  следует привязать действие, в котором будет вызываться экземпляр таймера TON:



**Примечание:** программист должен осторожно подходить к использованию классификаторов сохраняемых действий (S, DS, DS, SL), поскольку они могут вызывать трудно отслеживаемые побочные эффекты. Основная проблема, связанная с этими классификаторами – неявный вызов привязанного действия на шагах, предшествующих шагу, в котором это действие будет вызвано с классификатором R. Рекомендуется придерживаться следующего принципа: схема должна быть как можно более понятной и легкой в прочтении, даже если это приведет к увеличению ее размера.

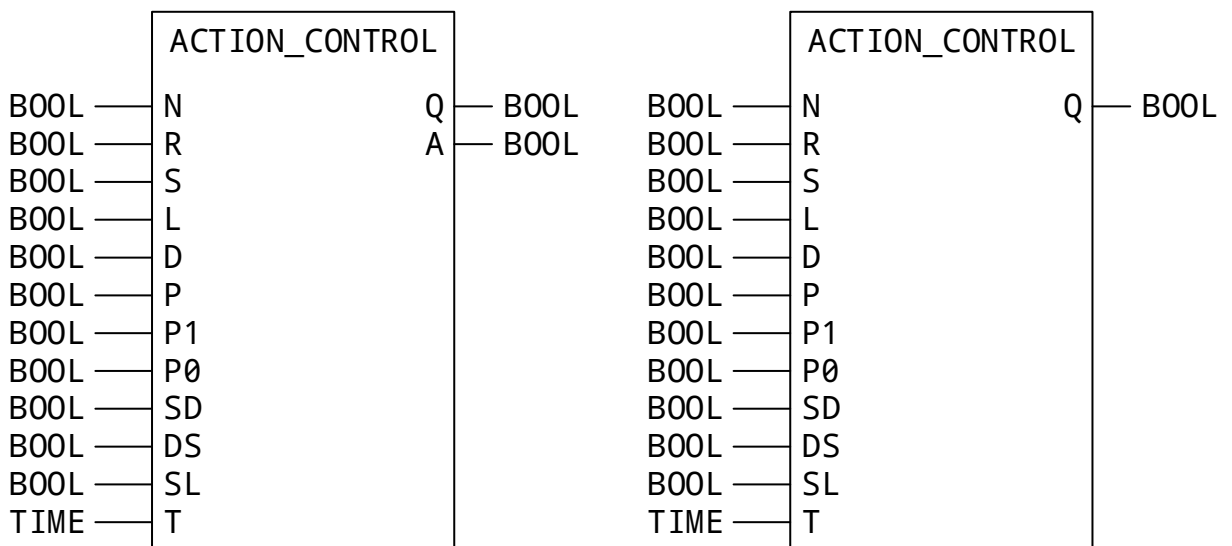
<sup>6</sup> Перед запуском таймер должен быть сброшен с помощью вызова с IN:=FALSE (прим. пер.)

### 3.6. Управление действиями

Этот раздел содержит информацию из 3-й редакции стандарта **МЭК 61131-3** (п. 6.7.4.6).

#### 3.6.1. Функциональный блок ACTION\_CONTROL

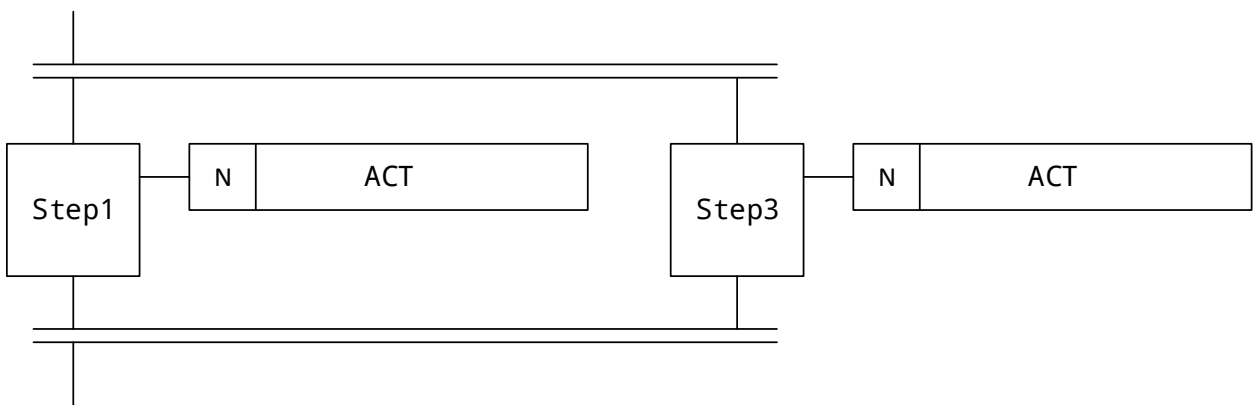
При анализе выполнения SFC-схемы необходимо учитывать, что выполнение действия не зависит от активности связанного с ним шага. При добавлении действия неявно создается экземпляр ФБ ACTION\_CONTROL, который управляет выполнением этого действия. При активации шага, связанного с действием, происходит вызов этого ФБ. Далее ФБ выполняется независимо от того, какой шаг является активным.



а) С «последним сканированием»

б) Без «последнего сканирования»

Следует отметить, что одно действие может быть связано с несколькими шагами и иметь различные классификаторы, но в каждом цикле ПЛК оно выполняется только однократно. Например, на приведенном ниже рисунке изображены параллельные шаги Step1 и Step3, к каждому из которых привязано действие АСТ. При активации параллельной ветви оба шага станут активными, но действие АСТ в каждом цикле ПЛК будет выполняться только один раз.





В большинстве случаев обработка каждого прохода SFC-схемы происходит следующим образом:

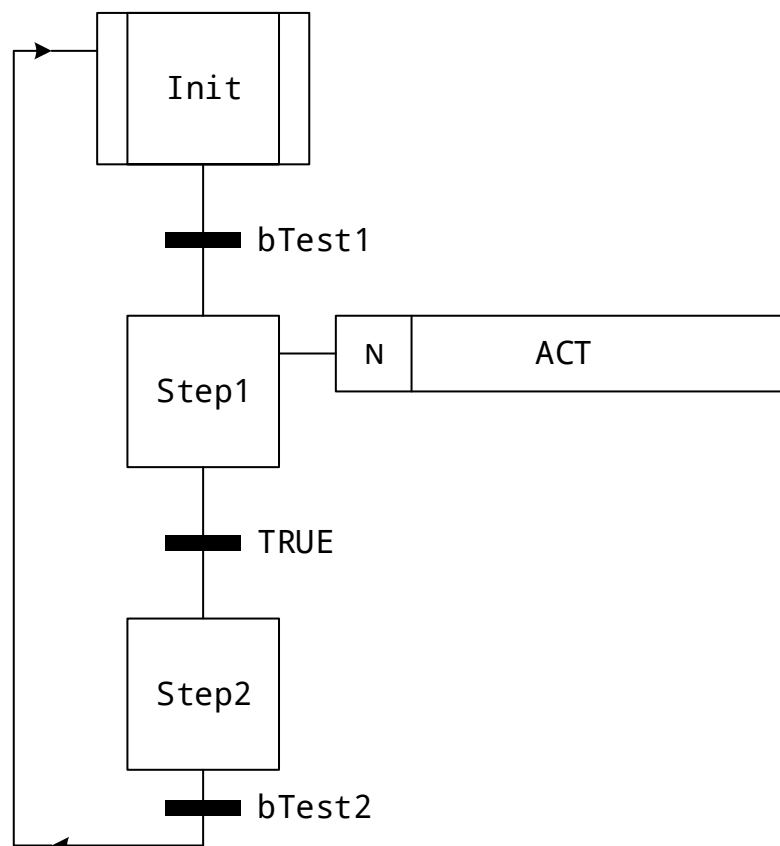
- выполняется сброс всех экземпляров ФБ ACTION\_CONTROL;
- происходит анализ состояний шагов из предыдущего прохода и выполнения условий переходов в текущем проходе;
- на основании анализа происходит активация новых шагов с записью информации в их экземпляры ФБ ACTION\_CONTROL;
- производится вызов экземпляров ФБ ACTION\_CONTROL;
- в зависимости от выходов экземпляров ФБ ACTION\_CONTROL происходит выполнение тех или иных действий (возможно, с последним сканированием – см. подробнее в [п. 3.6.2](#)).

Реализация обработки SFC-схемы в рамках конкретной среды программирования оказывает существенное влияние на выполнение программы.

### 3.6.2. Последнее сканирование (final scan)

Стандарт **МЭК 61131-3** допускает (но не требует) при обработке действий так называемое «последнее сканирование». В этом случае действие, связанное с определенным шагом, дополнительно однократно выполняется после перехода к следующему шагу, который не связан с этим действием. С точки зрения реализации – действие выполняется по сигналу заднего фронта своего экземпляра ФБ ACTION\_CONTROL.

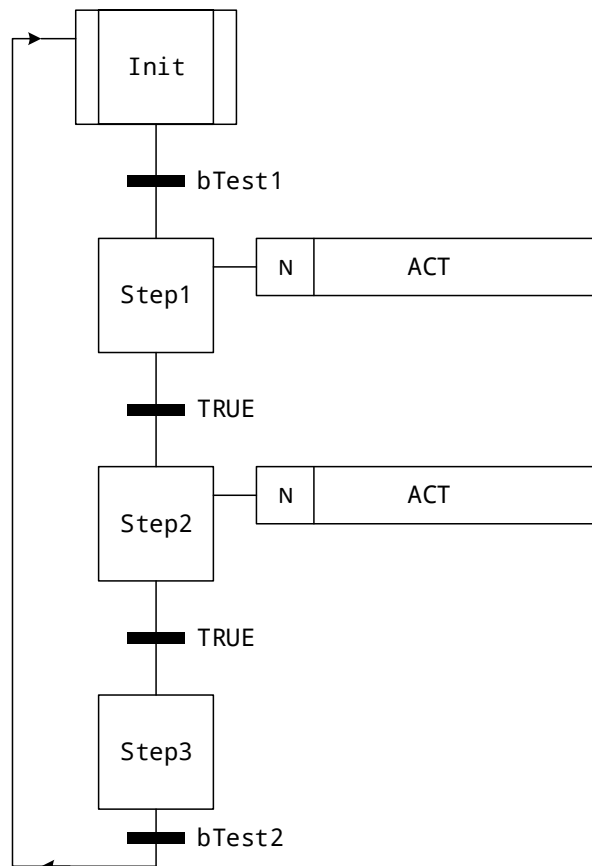
Поскольку вышесказанное сложно понять с первого раза, продемонстрируем этот эффект на примере:



Когда переменная `bTest1` примет значение `TRUE` – произойдет переход к шагу `Step1` и однократно выполнится действие `ACT`. В следующем цикле ПЛК будет проверено условие перехода к шагу `Step2` – оно всегда является истинным, так как определено через литерал `TRUE`. Отметим, что к шагу `Step2` действие `ACT` уже не привязано. При детектировании этих фактов действие `ACT` будет выполнено еще один раз, после чего произойдет переход к шагу `Step2`. Далее до конца обработки схемы действие `ACT` выполняться уже не будет.

Если бы при обработке данной схемы последнее сканирование не использовалось, то действие `ACT` было бы выполнено только один раз.

Рассмотрим более сложный пример:



Когда переменная `bTest1` примет значение `TRUE` – произойдет переход к шагу `Step1` и однократно выполнится действие `ACT`. В следующем цикле ПЛК будет проверено условие перехода к шагу `Step2` – оно всегда является истинным, так как определено через литерал `TRUE`. Отметим, что к шагу `Step2`, как и к предыдущему шагу `Step1`, привязано действие `ACT`. После детектирования этих фактов произойдет переход к шагу `Step2`. В данном случае дополнительного выполнения действия не произойдет, так как оно связано с обоими шагами (и предыдущим, и новым), и экземпляр ФБ `ACTION_CONTROL` не получит сигнал заднего фронта. На шаге `Step2` действие `ACT` выполнится однократно. В следующем цикле ПЛК будет проверено условие перехода к шагу `Step3` – оно всегда является истинным, так как определено через литерал `TRUE`. Отметим, что к шагу `Step3` действие `ACT` уже не привязано. После детектирования этих фактов действие `ACT` будет выполнено еще один раз, после чего произойдет переход к шагу `Step3`. Далее до конца обработки схемы действие `ACT` выполняться уже не будет.

Таким образом, при проходе схемы действие АСТ будет выполнено три раза. Если бы при обработке данной схемы последнее сканирование не использовалось, то действие АСТ было бы выполнено два раза.

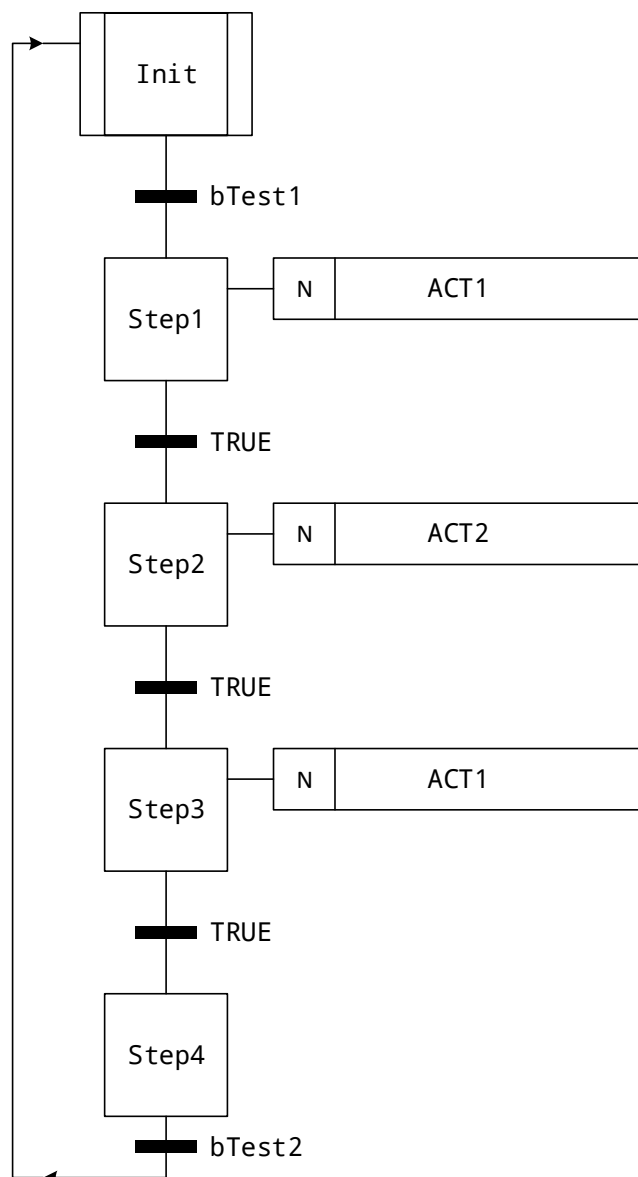
Теперь эффект последнего сканирования должен быть понятен – оно позволяет произвести дополнительное выполнение действия перед переходом к шагу, на котором это действие выполняться уже не должно.

Реализация обработки действия АСТ в примере выше может быть следующей:

```
ACTION ACT
  IF Step2.x OR Step1.x THEN
    // Выполнение циклических действий;
  ;
  ELSE
    // Выполнение завершающих действий;
  ;
  END_IF
END_ACTION
```

В некоторых реализациях SFC (расширяющих функционал стандарта МЭК) пользователю доступны специальные флаги, которые позволяют получить информацию о состоянии экземпляров ФБ ACTION\_CONTROL. Тогда обработка последнего сканирования может быть реализована без использования неявных переменных шагов (.X, .T).

При последнем сканировании может возникнуть ситуация, когда два действия выполняются в пределах одного цикла ПЛК, даже если на схеме нет параллельных ветвей. Рассмотрим эту проблему на примере следующей SFC-схемы:



Реализация действий:

```

ACTION ACT1
    x := x + 1;
END_ACTION
  
```

```

ACTION ACT2
    y := x;
END_ACTION
  
```

В таблице приведена циклограмма обработки схемы после перехода к шагу Step1:

Цикл ПЛК	Step1	Step2	Step3	ACT1	ACT2	x	y
1	активен	неактивен	неактивен	выполняется	не выполняется	1	0
2	неактивен	активен	неактивен	выполняется (ПС <sup>7</sup> )	выполняется	2	1 или 2
3	неактивен	неактивен	активен	выполняется	выполняется (ПС <sup>3</sup> )	3	2 или 3
4	неактивен	неактивен	неактивен	выполняется (ПС <sup>3</sup> )	не выполняется	4	2 или 3

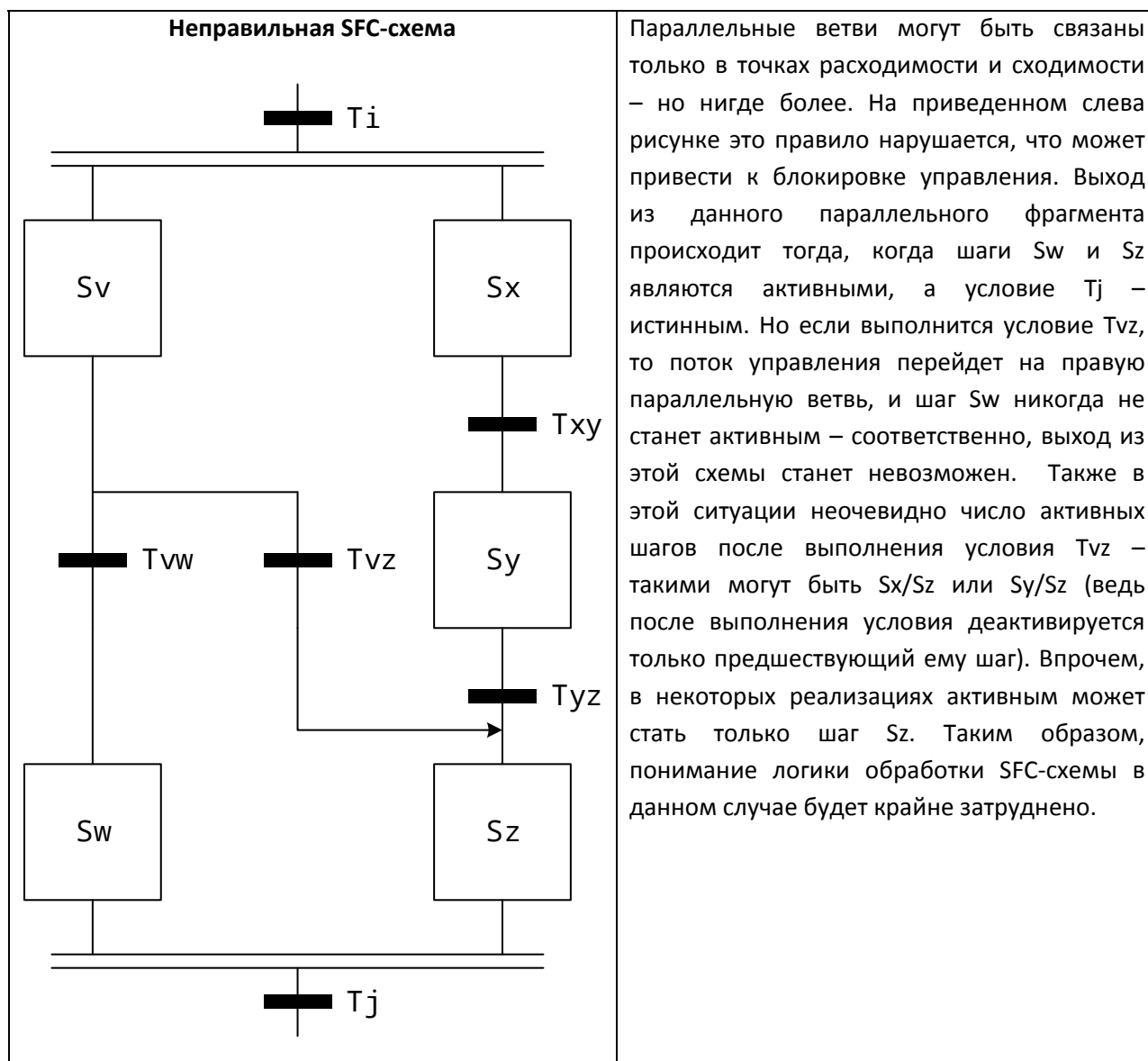
Таким образом, в циклах 2 и 3 выполняются оба действия (ACT1 и ACT2) – что может являться неожиданностью для разработчика. Более того, стандарт не определяет порядок выполнения действий в этом случае – так что если они работают с общими данными, то результат вычислений может зависеть от реализации конкретной среды программирования. Поэтому в циклах 2-4 для переменной **y** указано два возможных значения: первое из них соответствует порядку выполнения ACT1/ACT2, а второе – ACT2/ACT1.

<sup>7</sup> Выполняется при последнем сканировании (прим. пер.)

## 4. Рекомендации по использованию SFC

### 4.1. Расходимость и сходимость

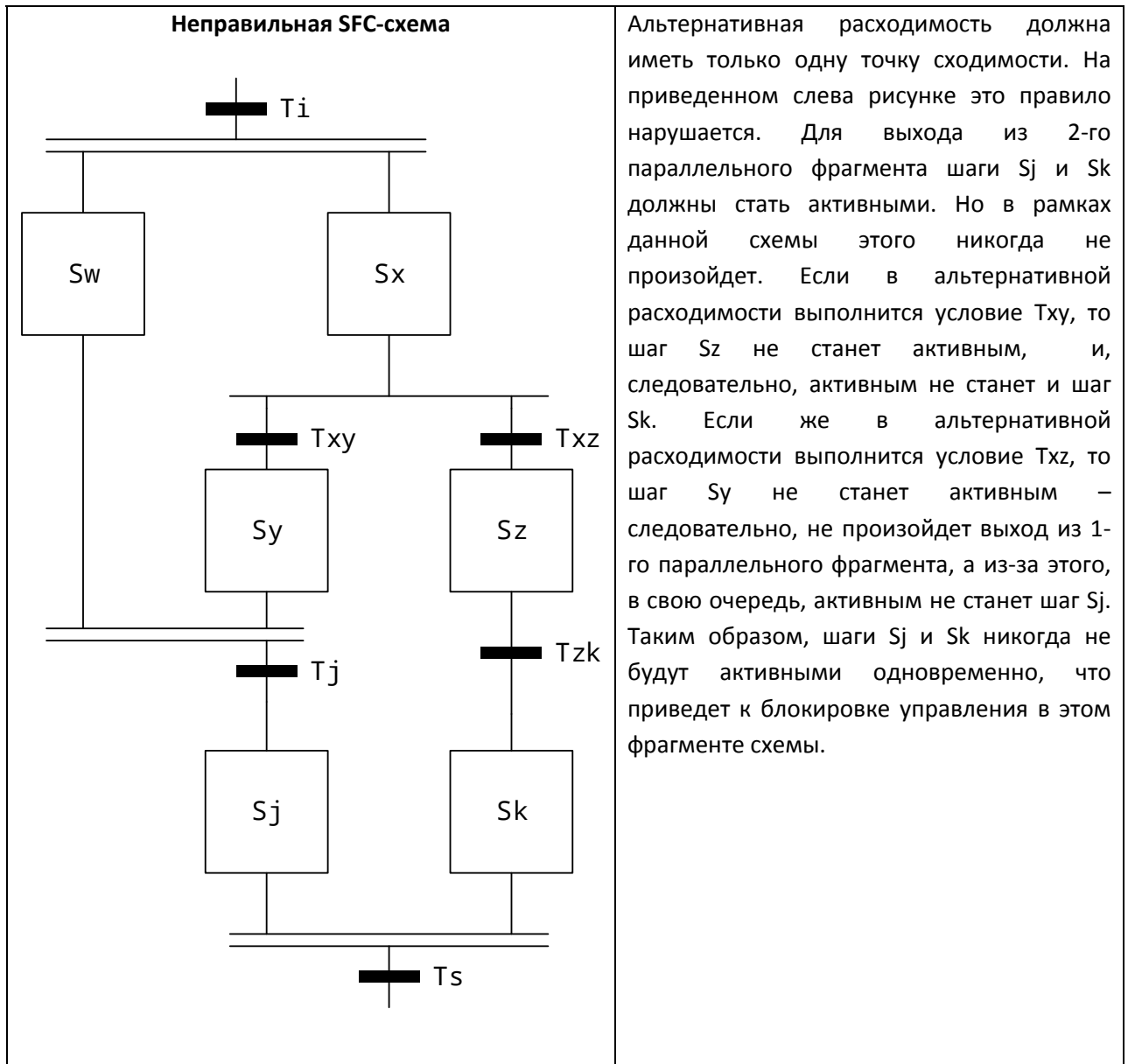
Параллельные ветви могут быть связаны только в точках расходимости и сходимости



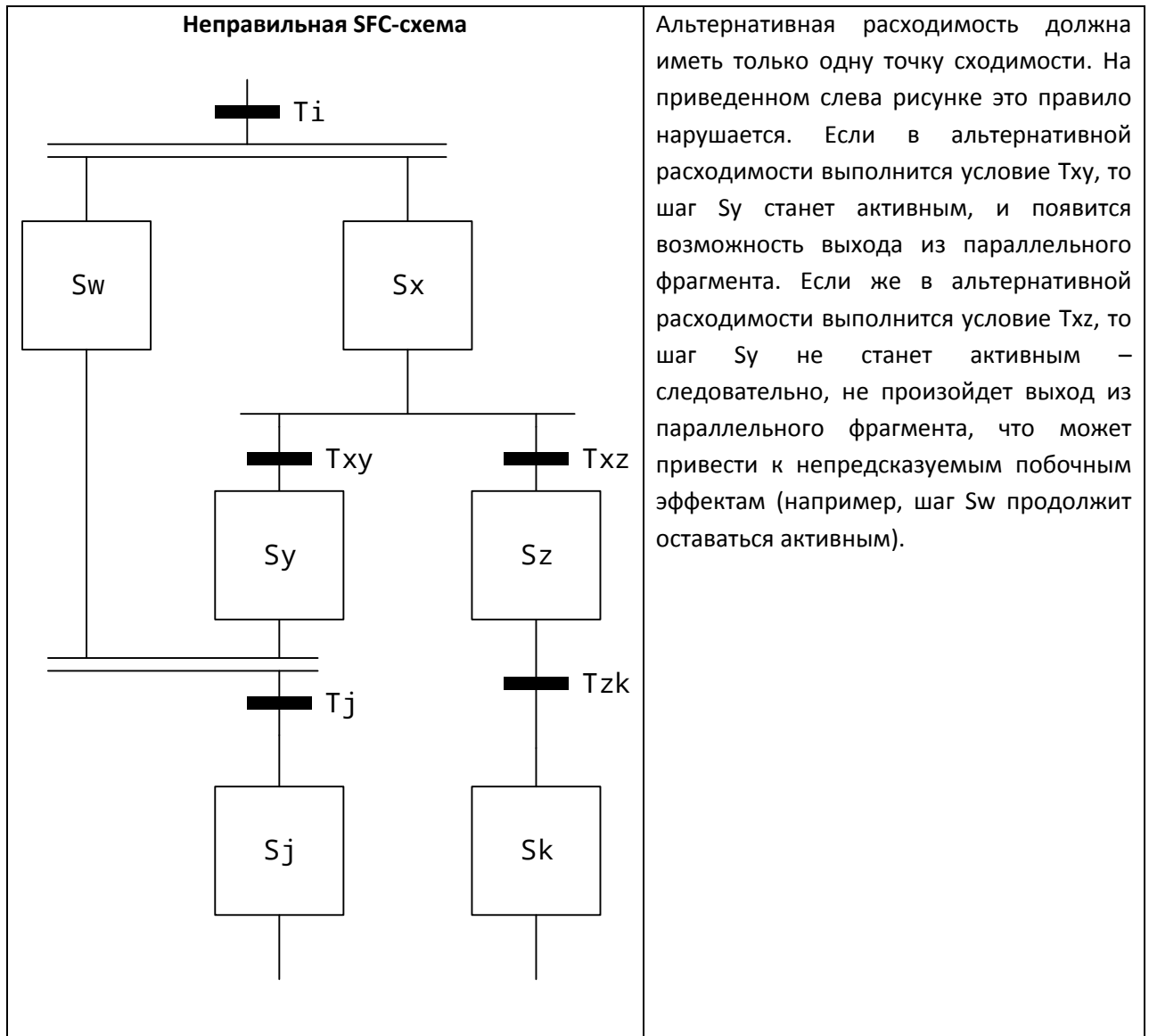
**Каждый параллельный фрагмент SFC-схемы должен иметь свою точку сходимости**



## Альтернативная расходимость должна иметь только одну точку сходимости (1)



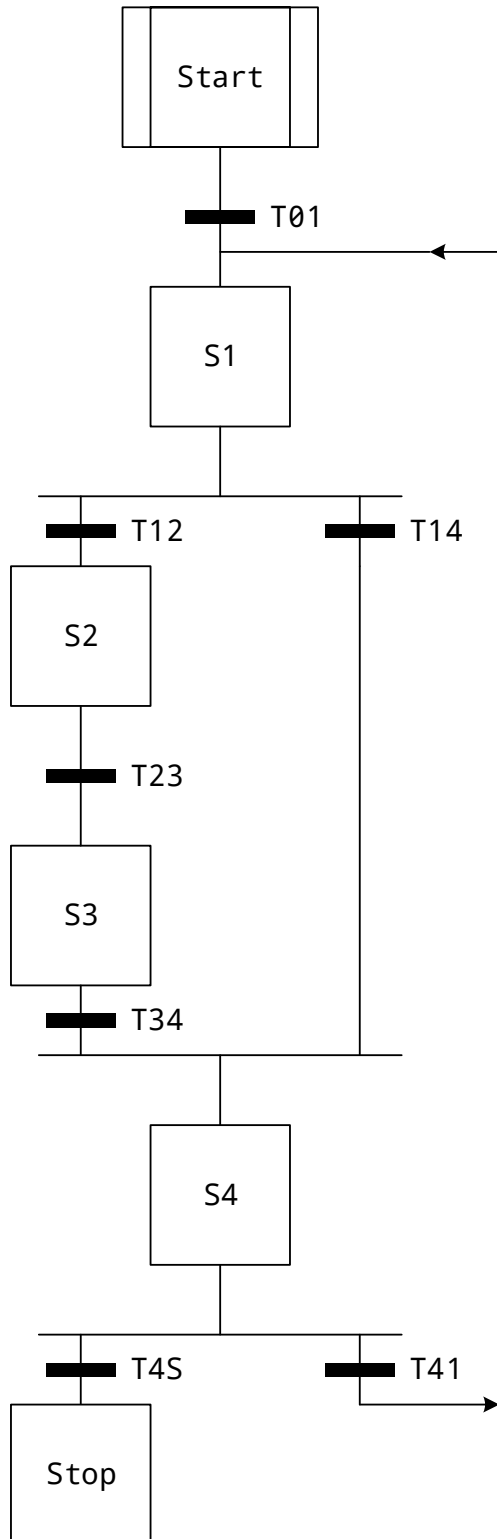
## Альтернативная расходимость должна иметь только одну точку сходимости (2)



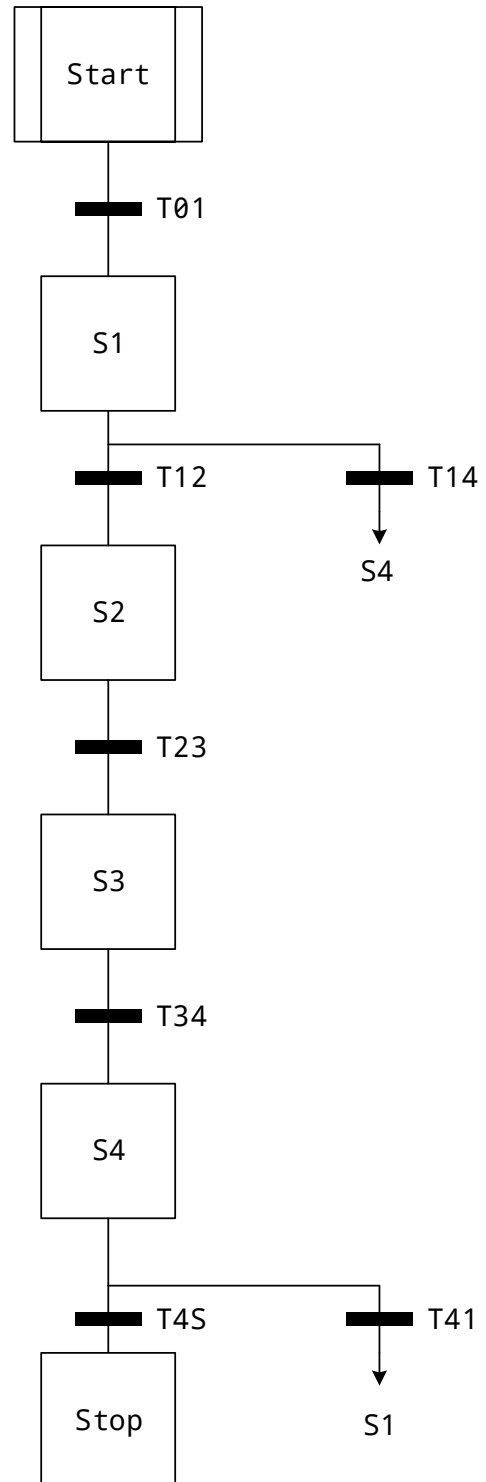


## 4.2. Линеаризация в SFC

В SFC-схемах нет стрелок, указывающих направление переходов. Предполагается, что схема обрабатывается последовательно сверху вниз. Единственным исключением являются *циклы*, в которых возможен возврат к расположенным выше шагам.



а) Стандартный SFC-синтаксис



б) Использование примитива Jump

В некоторых средах программирования поддержан элемент **JUMP**, который является расширением стандарта МЭК 61131-3. Этот элемент представлен в виде символа  $\downarrow$ , расположенного после условия перехода, и содержит имя шага, к которому произойдет переход. Выше изображена схема со стандартным изображением циклов (рис. а) и эквивалентная ей схема с операторами JUMP (рис. б). Использование этих операторов может повысить читабельность схем. Так как в SFC запрещены пересечения линий, то в некоторых случаях использование JUMP остается единственным способом изобразить нужную последовательность переходов.

### 4.3. Условия перехода

В SFC-схемах переходы между шагами происходят при выполнении условий. Крайне рекомендуется использовать условия перехода, которые взаимно исключают друг друга – это позволяет избежать неоднозначных ситуаций при обработке схемы. Создание условий, которые могут выполняться одновременно («пересекающихся» условий), является рискованным выбором профессионала или, что более вероятно, ошибкой новичка. Настоятельно советуем избегать такого подхода и всегда использовать только взаимоисключающие условия.

Стандарт МЭК 61131-3 подразумевает, что в каждый момент времени только одна ветвь альтернативной расходимости будет активной. Если выполняются условия для нескольких альтернативных ветвей, то при отсутствии других факторов активной станет самая левая ветвь. В некоторых средах разработки программист может назначать для альтернативных ветвей приоритеты выполнения.

Концепция приоритетов не является панацеей и имеет свои недостатки. Во-первых, само понятие «приоритет» может ввести пользователя в заблуждение, заставив думать, что в рассматриваемой ситуации будут выполнены все альтернативные ветви, а приоритет определит порядок их выполнения. Во-вторых, крайне велика вероятность ошибки при использовании приоритетов для условий расходимости – разобраться в тонкостях обработки сложной схемы становится очень затруднительно. Поэтому многие авторы (и мы в их числе) рекомендуют избегать использования приоритетов и применять только взаимоисключающие условия.

Для демонстрации недостатков системы приоритетов рассмотрим следующую SFC-схему с пересекающимися условиями (где  $v1, v2, v3$  – переменные POU):

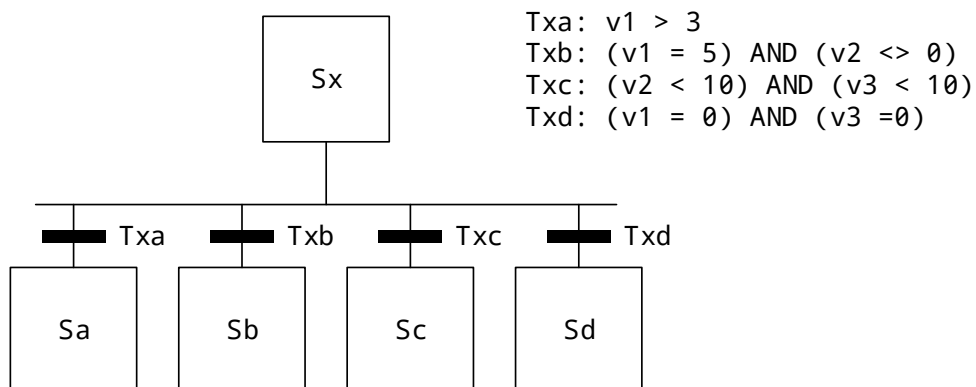


Рис. 5. Альтернативная расходимость с пересекающимися условиями

Пусть условия имеют следующие приоритеты:

*Txa важнее Txb и Txc*  
*Txb важнее Txc*  
*Txb важнее Txd*

Эту систему удобно представить в виде [направленного графа](#), каждый узел которого соответствует одному из условий (см. рис. 6а). Переходы по графу будут соответствовать понижению приоритета. Таким образом, довольно легко определить числовые значения для приоритетов условий (чем больше значение, тем ниже приоритет):

*Приоритет Txd = 4*  
*Приоритет Txc = 3*  
*Приоритет Txb = 2*  
*Приоритет Txa = 1*

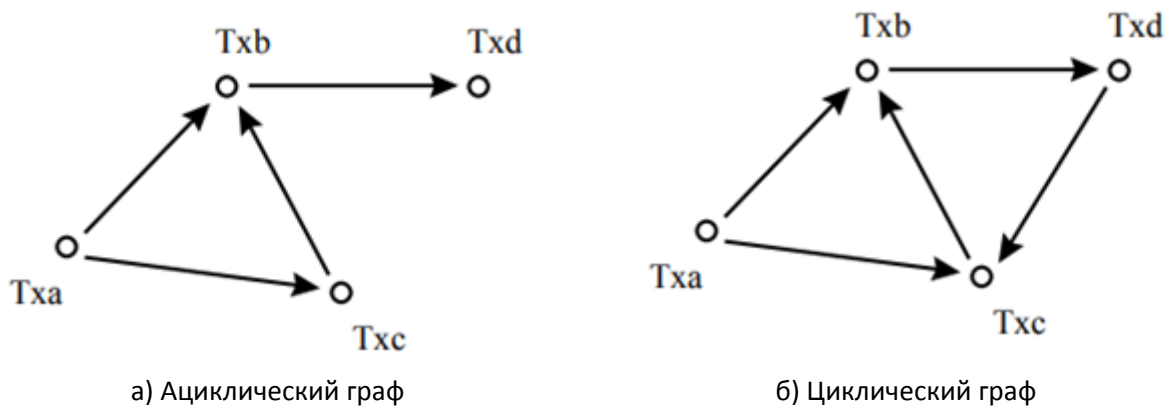


Рис. 6. Направленный граф приоритетов условий

Метод направленных графов удобно использовать в тех случаях, когда приоритеты не могут быть изначально заданы в виде числовых значений. Представленный на рис. 6а является [ациклическим](#) (разомкнутым); это является ключевым условием возможности определения значений приоритетов. На рис. 6б изображен [циклический граф](#), который получен из ациклического при добавлении нового условия:

*Txd важнее Txc*

Циклический граф является замкнутым. Для циклического графа невозможно определить набор значений приоритетов, удовлетворяющий всем условиям. Единственный возможный выход – изменить условия таким образом, чтобы перейти от циклического графа к ациклическому. Таким образом, условия станут взаимоисключающими.

Рассмотрим простейший пример: пусть условия  $T_x$  и  $T_y$  являются пересекающимися. На рис. 7а приведена [диаграмма Венна](#), где общая область обоих условий отмечена как  $T_{xy}$ . Эту область можно интерпретировать как ситуацию, в которой оба условия ( $T_x$  и  $T_y$ ) являются истинными. Если мы хотим разделить условия (чтобы в каждый момент времени истинным являлось только одно из них) и при этом сделать условие  $T_x$  более приоритетным, то следует переопределить области следующим образом (новые области будут называться  $T_x'$  и  $T_y'$ ):

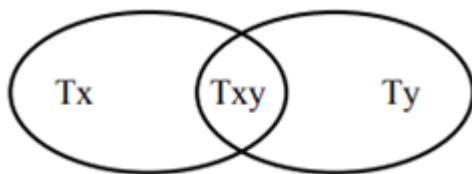
$$\text{Область } (T_x') = \text{Область } (T_x) - \text{Область } (T_y)$$

$$\text{Область } (T_y') = \text{Область } (T_y)$$

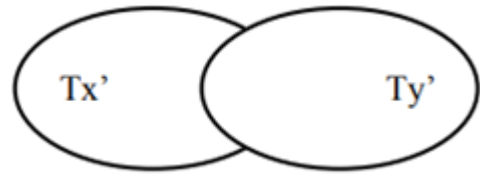
Графическое представление новой системы условий приведено на рис. 7б.

Довольно просто описать отношения условий с помощью булевой алгебры:

$$T_x' = T_x \text{ AND NOT } T_y$$



а) Пересекающиеся условия



б) Взаимоисключающие условия

Рис. 7. Диаграмма Венна для двух условий

В ситуациях с большим числом условий переход от пересекающихся к взаимоисключающим условиям происходит аналогично. Следует отметить, что при увеличении числа условий экспоненциально увеличивается сложность описывающей их системы выражений. На рис. 8а приведена диаграмма Венна для трех пересекающихся условий. На диаграмме присутствует 4 области пересечения –  $T_{xy}$ ,  $T_{xz}$ ,  $T_{yz}$  и  $T_{xyz}$ . Определим следующие приоритеты:

*$T_y$  важнее  $T_x$  и  $T_z$*

*$T_x$  важнее  $T_z$*

Переопределим области условий:

$$\text{Область } (T_x') = \text{Область } (T_x) - \text{Область } (T_{xy}) - \text{Область } (T_{xyz})$$

$$\text{Область } (T_y') = \text{Область } (T_y)$$

$$\text{Область } (T_z') = \text{Область } (T_z) - \text{Область } (T_{xz}) - \text{Область } (T_{yz}) - \text{Область } (T_{xyz})$$

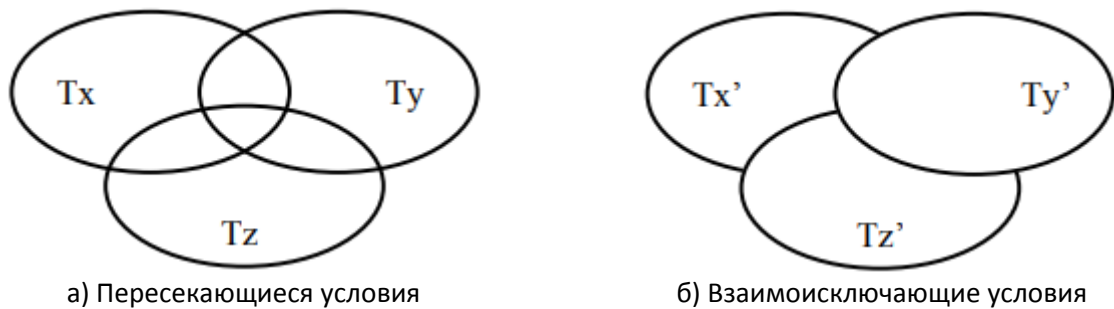


Рис. 8. Диаграмма Венна для трех условий

Теперь можно описать отношения условий с помощью булевой алгебры:

$$Tx' = Tx \text{ AND NOT } Ty$$

$$Tz' = Tz \text{ AND NOT } (Tx \text{ OR } Ty)$$

В заключении вернемся к схеме, изображенной на [рис. 5](#), и отредактируем ее условия, сделав их взаимоисключающими:

$$Txa' = Txa \text{ AND NOT } Txb$$

$$Txb' = Txb$$

$$Txc' = Txc \text{ AND NOT } (Txa \text{ OR } Txb \text{ OR } Txd)$$

$$Txd' = Txd \text{ AND NOT } (Txa \text{ OR } Txb)$$

Так как мы знаем формулировки условий, то можем подставить их в полученные выражения:

$$Txa' = (v1 < 3) \text{ AND NOT } (v1 = 5 \text{ AND } v2 < > 0)$$

$$Txb' = (v1 = 5 \text{ AND } v2 < > 0)$$

$$Txc' = (v2 < 10 \text{ OR } v3 < 10) \text{ AND NOT } (v1 > 3 \text{ OR } (v1 = 5 \text{ AND } v2 < > 0) \text{ OR } (v1 = 0 \text{ AND } v3 = 0))$$

$$Txd' = (v1 = 0 \text{ AND } v3 = 0) \text{ AND NOT } (v1 > 3 \text{ OR } (v1 = 5 \text{ AND } v2 < > 0))$$

Приведенные выше примеры демонстрируют, что даже использование приоритетов затрудняет понимание схемы, содержащей пересекающиеся условия. Это позволяет нам сформулировать [правило 4.4](#).

#### 4.4. Не используйте пересекающиеся условия

Не используйте пересекающиеся условия – они усложняют чтение схемы и, кроме того, невозможно предсказать, каким образом они будут обрабатываться в рамках конкретного ПО. Даже наличие системы приоритетов не сильно снижает вероятность ошибки – особенно в случае присутствия в схеме расходимостей. Разобраться в тонкостях обработки такой схемы становится очень затруднительно. Поэтому многие авторы (и мы в их числе) рекомендуют избегать использования приоритетов и применять только взаимоисключающие условия.

#### 4.5. Зависимость от предыдущего состояния

В некоторых ситуациях необходимо, чтобы действия, связанные с текущим шагом, выполняли разные операции в зависимости от того, какой из шагов был активен при прошлом проходе схемы. В рассмотренных ранее примерах действия выполнялись в одном из следующих случаев:

- однократно при активации связанного с ними шага;
- пока связанный с ними шаг был активен;
- в течение заданного времени.

Мы не рассматривали вариант, при котором поведение действий определялось тем, какой из шагов схемы был активен до перехода к текущему шагу. Тем не менее, достаточно часто требуется именно такой способ обработки действий.

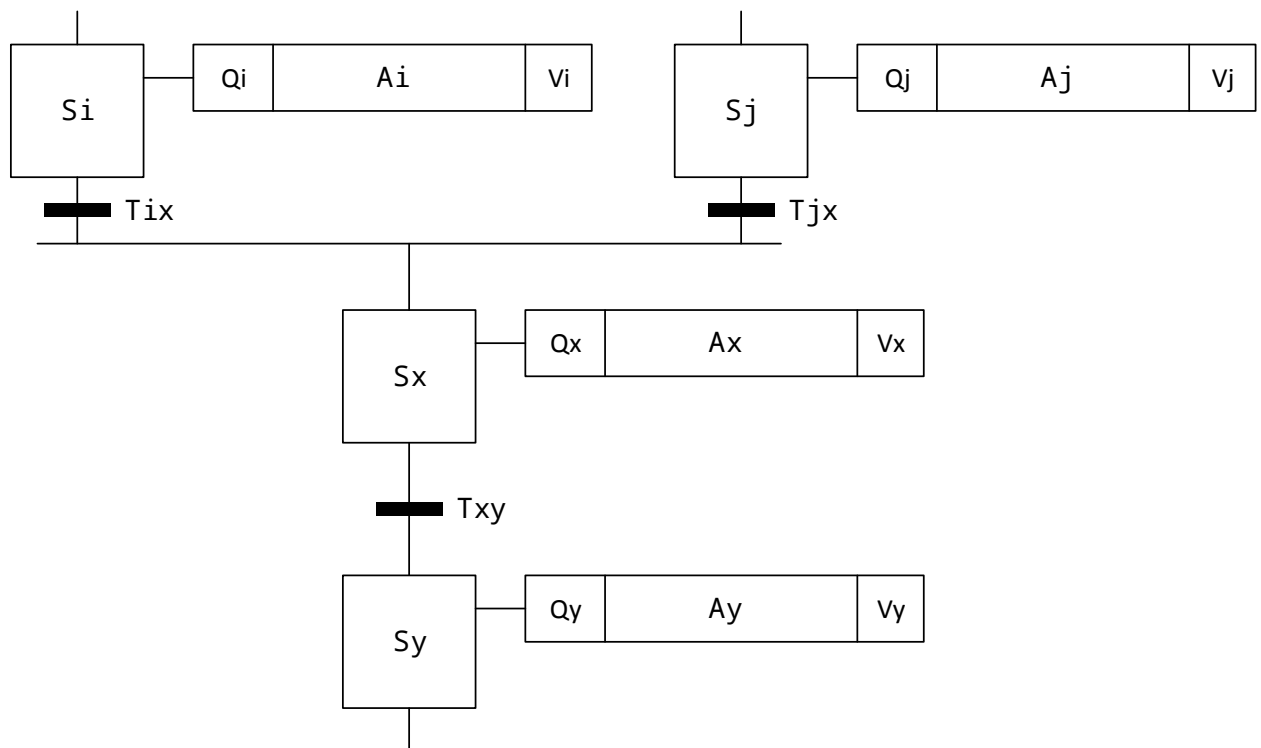


Рис. 9. Схема с зависимостью от предыдущего состояния

На рис. 9 приведен пример с действием Ax, которое выполняет разные операции в зависимости от того, какой из шагов схемы был активен при предыдущем проходе – Si или Sj. Если зависимость от предыдущих шагов незначительно влияет на совершаемые операции (т.е. основная часть кода является общей для обоих случаев), то проще всего организовать зависимость через дополнительные переменные-флаги (см. рис. 10). В зависимости от того, какой из шагов был активен ранее (Si или Sj), связанное с ним действие (Ai или Aj) активирует соответствующий флаг (FromSi или FromSj). После выполнения действия Ax флаги сбрасываются в FALSE.

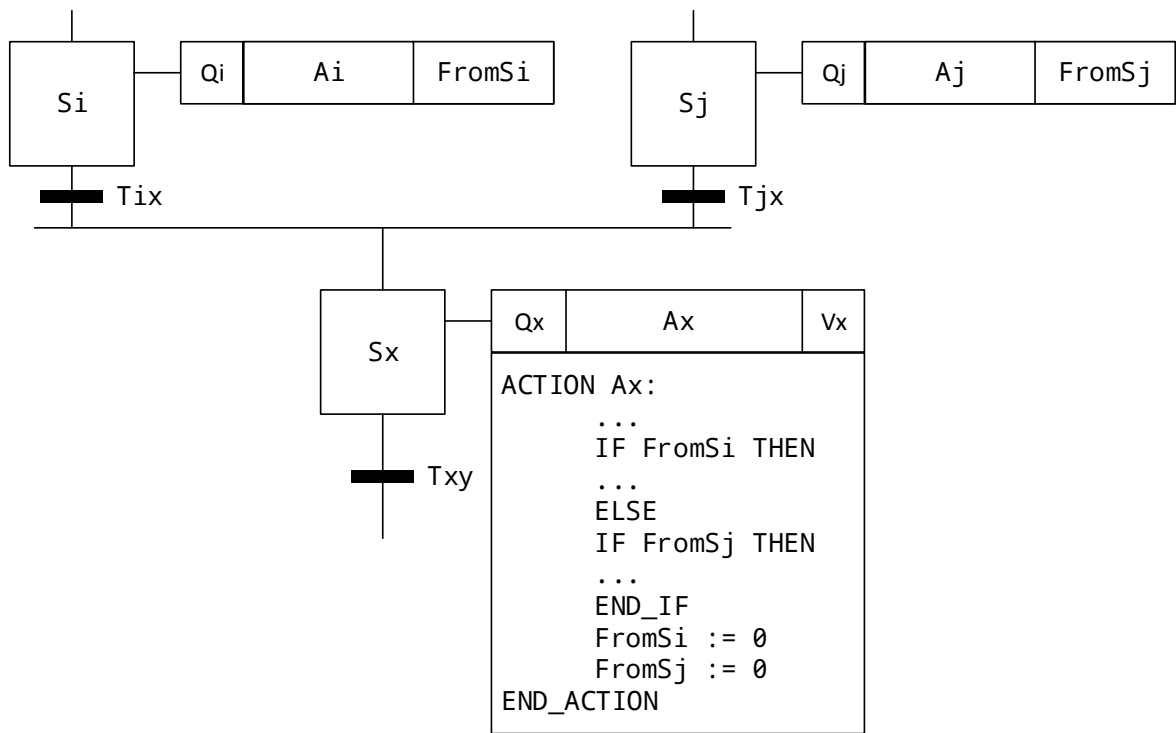


Рис. 10. Реализация зависимости от предыдущего состояния через флаги

Если же в зависимости от предыдущих шагов действие должно выполнять совершенно разные операции, то корректнее будет разделить связанный с ним шаг на два отдельных шага, а само действие – на два отдельных действия (см. рис. 11). Вариант с разделением шагов и действий имеет значительное преимущество перед вариантом с флагами – так как в этом случае зависимость действия от предыдущих шагов становится очевидной при ознакомлении со схемой. В противном случае (см. рис. 10) для понимания зависимости следует изучить код действия.

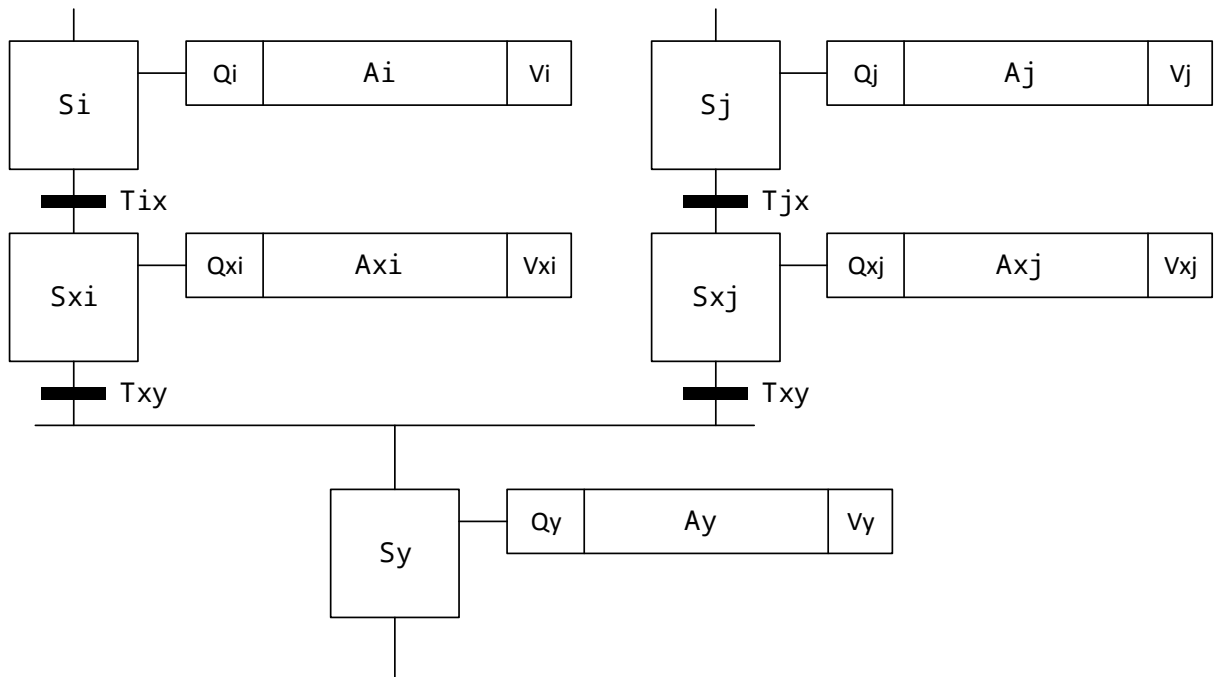


Рис. 11. Реализация зависимости от предыдущего состояния через разделение шагов и действий

#### 4.6. Параллельные ветви

Синхронизировано выполняемые шаги позволяют избежать применения сохраняемых действий, а в некоторых случаях – организовать взаимодействие между параллельными ветвями. Например, это позволяет повысить читабельность схемы, действия которой имеют классификаторы Set (S) и Reset (R). Как мы уже упоминали в [п. 3.5.1](#), можно отказаться от этих классификаторов перейдя к эквивалентной схеме, в которой каждый шаг, на котором должно выполняться заданное действие, связан с этим действием, отмеченным классификатором N. Недостатком такого решения является увеличение количества действий на схеме, что может затруднить ее восприятие. Более изящным решением является переход к параллельным ветвям.

На рис. 12а приведен пример схемы с классификаторами S и R, в которой действие Axy выполняется на шагах Sx и Sy. Предположим, что шаги могут содержать дополнительные уникальные действия – иначе можно было бы объединить их в шаг Sxy, к которому было бы привязано действие Axy с классификатором N. Таким образом, вариант с объединением шагов в данном случае придется отбросить.

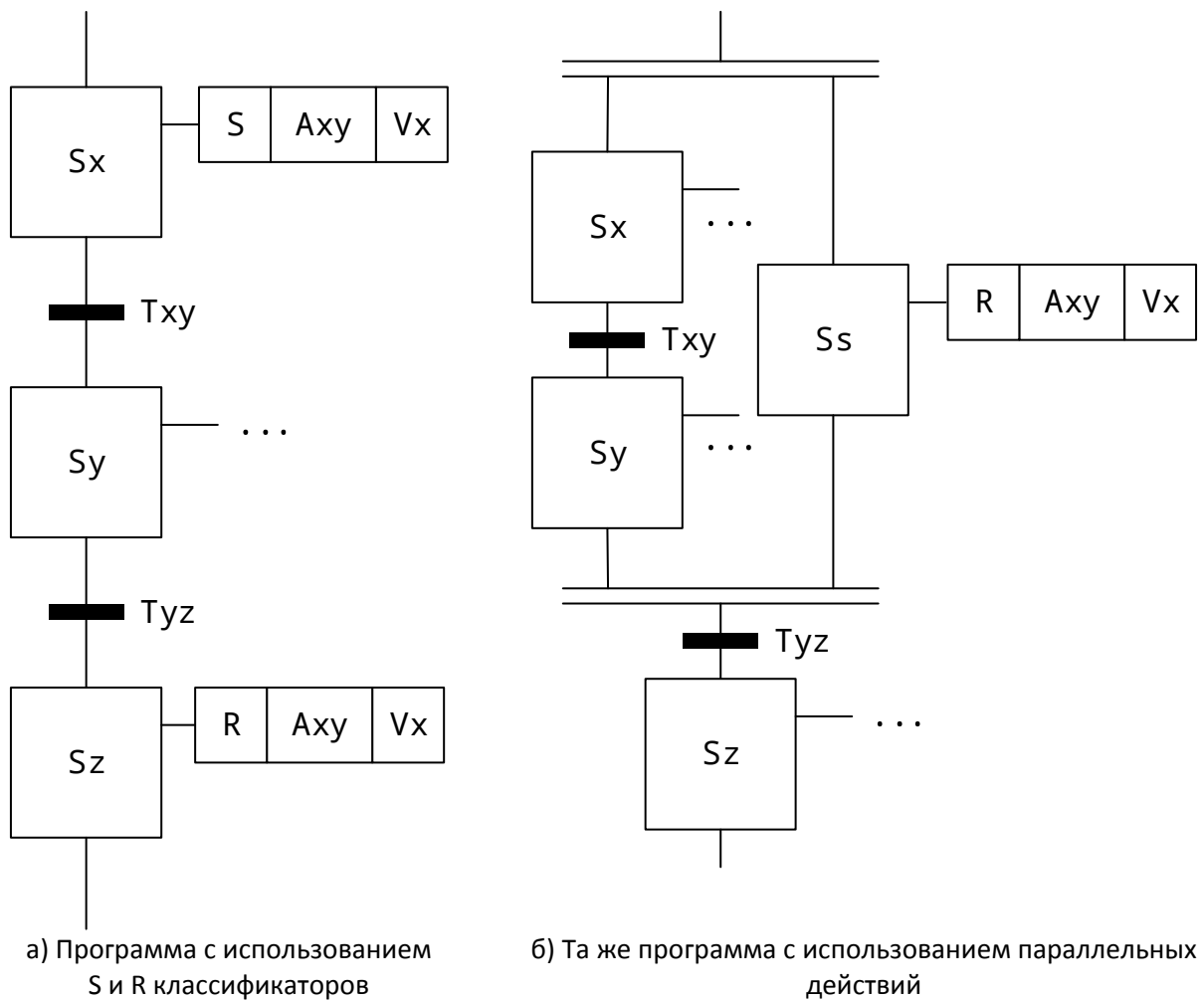


Рис. 12. Использование параллельных ветвей для повышения читабельности схемы



На рис. 12b изображена схема, эквивалентная схеме с рис. 12a. Вместо действий с классификаторами S и R в ней добавлена параллельная ветвь с шагом Ss, к которому привязано действие Axy с классификатором N. Условие выхода из параллельной расхожимости совпадает с условием перехода к шагу сброса действия на исходной схеме.

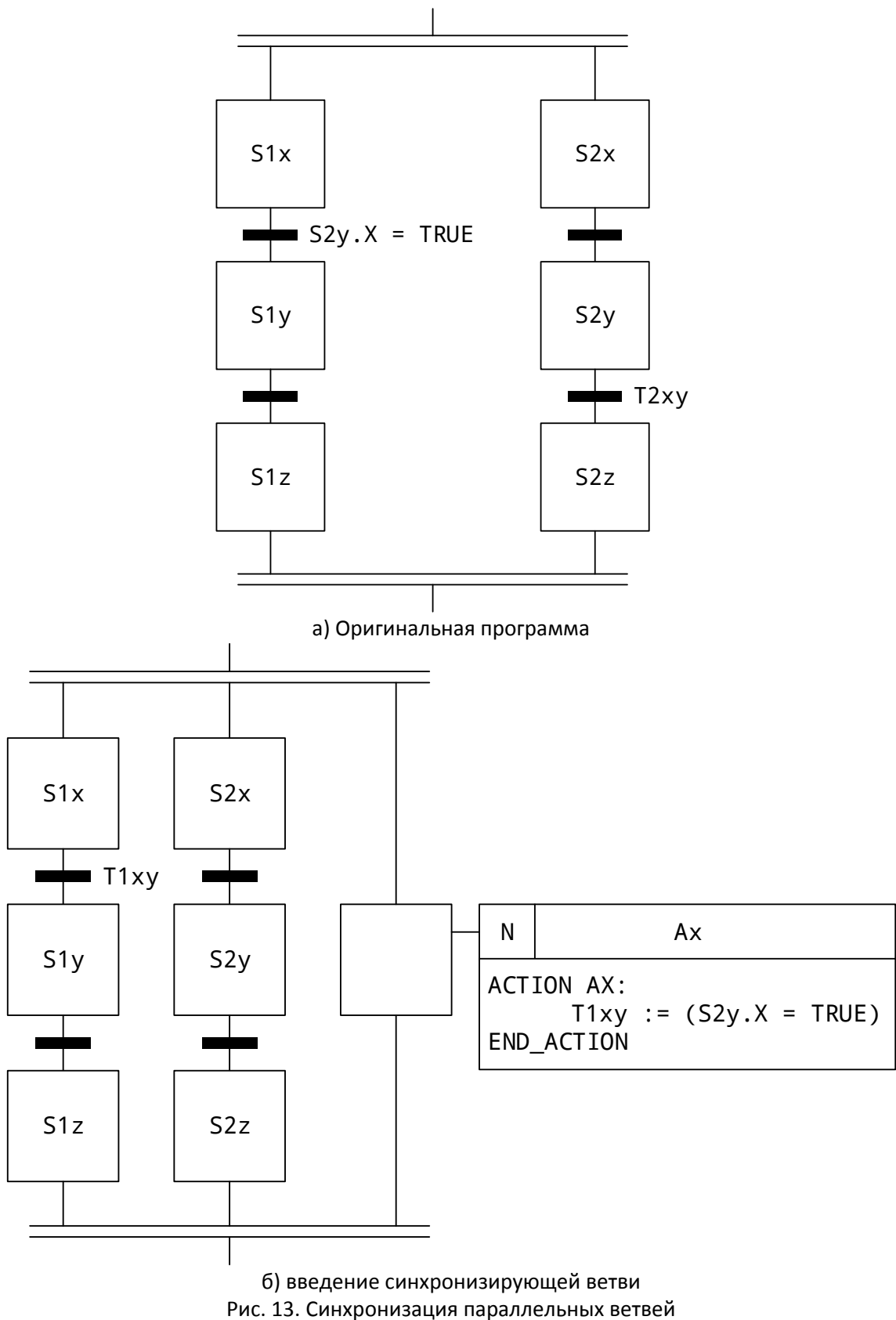
Рассмотренный подход позволяет также упростить схемы, содержащие действия с классификаторами SD, DS и SL. Выполнение этих действий зависит от времени, прошедшего с момента активации начального для них шага. Поэтому на шаге Sx должен быть запущен таймер, который будет проверяться на остальных шагах. Это установит сильную и неявную связь между действиями, которая станет понятна только после изучения их кода (напомним, код действий может не отображаться в редакторе SFC и быть доступен только в отдельных окнах среды программирования). Таким образом, повторное использование кода действий будет затруднено.

Но проанализировав эквивалентные схемы с параллельными ветвями для классификаторов SD, DS и SL можно заметить, что все они имеют одинаковую структуру (см. рис. 12b). Отличие заключается лишь в методе обработки таймера активности шага Sx:

- SD – в каждом цикле ПЛК в действии Axy время, прошедшее с момента активации шага Sx (заметим, что оно совпадает с моментом активации шага Ss), сравнивается с заданной уставкой. Это время можно определить как сумму неявных переменных шагов параллельной ветви:  $Sx.T + Sy.T$ . Если эта величина превышает значение уставки, то начинается выполнение основного кода действия;
- DS – обработка аналогична предыдущему случаю, но контролируется исключительно время активности начального шага ( $Sx.T$ );
- SL – обработка похожа на вариант с SD, но код действия начинает выполняться сразу после активации шага Sx и прекращает после того, как прошедшее время ( $Sx.T + Sy.T$ ) превысит значение уставки.

Описанные выше варианты могут комбинироваться – например, если потребуется начать выполнение действия через время  $t1$  после активации шага Sx и выполнять его в течение заданного времени  $t2$ . Хотя такое поведение сложно организовать в рамках стандарта, стоит отметить, что его обработка все равно будет происходить в соответствии с рис. 12b.

В рассмотренном выше примере шаг Ss контролирует состояния всех остальных шагов (в частности, время их активности). Это позволяет отказаться от добавления дополнительного кода в действия этих шагов, что, в свою очередь, облегчает их повторное использование в схеме. Управляющее действие Axy также может тиражироваться. Общий принцип контроля шагов в случае параллельной расхожимости приведен на рис. 13:



На рис. 13а изображены две параллельные ветви. Активация шага S1y первой ветви происходит при деактивации шага S2y второй ветви. Добавление управляющей ветви с действием Ax (рис. 13б) упрощает схему, позволяя выделить весь код управления переходами в один программный модуль.

#### 4.7. Независимость действий

Читабельность схемы и возможность повторного использования кода действий является важным условием для создания хорошей SFC-программы. Не менее важным является тщательный анализ кода действий на предмет его корректного выполнения. При этом следует помнить, что в пределах цикла ПЛК может быть активно несколько шагов схемы, и с каждым из этих шагов могут быть связаны несколько действий.

Ключевым условием корректности действий, выполняемых в одном цикле ПЛК, является их независимость друг от друга. Код каждого действия не должен использовать результаты выполнения кода других действий или изменять переменные, считываемые в них. Это связано с тем, что стандарт МЭК не определяет порядок выполнения действий в пределах цикла ПЛК. В различных средах разработки определение порядка выполнения может быть реализовано по-разному, и в большинстве случаев конкретный вариант неизвестен программисту.

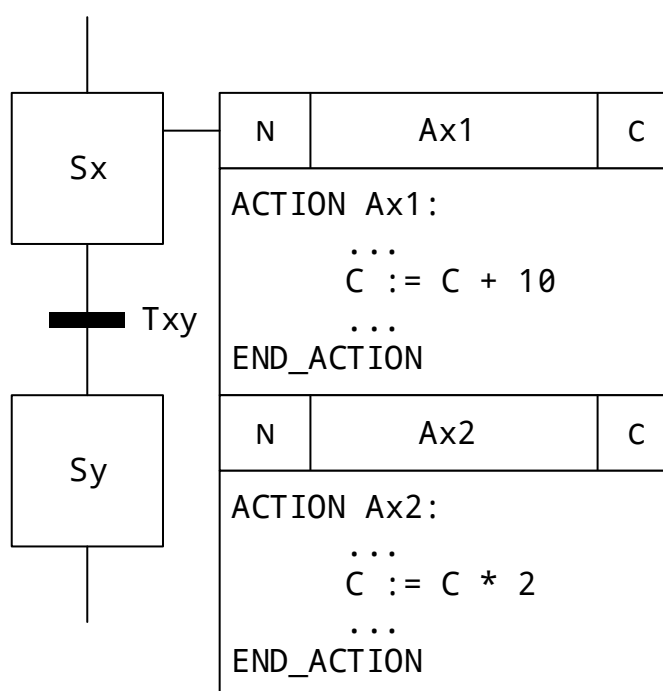


Рис. 14. Пример зависимых действий

Поэтому следует считать, что в пределах цикла ПЛК действия выполняются параллельно. Их код должен быть реализован таким образом, чтобы на него не влиял порядок выполнения действий. На рис. 14 это правило нарушается. Если предположить, что в начале цикла  $C=5$ , то в зависимости от порядка выполнения результирующее значение может быть равно как 30 (Ax1-Ax2), так и 20 (Ax2-Ax1).

Можно предположить, что ситуация, изображенная рис. 14, является последствием несогласованной работы нескольких программистов – один из них занимался реализацией действия  $Ax_1$ , другой – реализацией действия  $Ax_2$ . При внедрении готового кода в проект не был проведен анализ зависимостей между действиями. В некоторых средах программирования такие проблемы могут детектироваться с помощью встроенных средств: перекрестных ссылок, сообщений компилятора и т.д. Но несмотря на это рекомендуется избегать зависимости действий друг от друга.

Пусть  $Ax$  и  $Ay$  – действия, выполняемые в рамках одного цикла. Независимыми эти действия будут только тогда, когда выполняются все 3 условия, приведенные ниже:

1. Ни одна из переменных, которые записываются в  $Ax$ , не записывается в  $Ay$ .
2. Ни одна из переменных, которые записываются в  $Ax$ , не считывается в  $Ay$ .
3. Ни одна из переменных, которые записываются в  $Ay$ , не считывается в  $Ax$ .

Отметим, что условие допускает считывание в коде различных действий одних и тех же переменных.

Нарушение условия (1) соответствует случаю, когда различные действия производят запись значения в одну и ту же переменную. Если это не является ошибкой программирования, то свидетельствует о плохой связности действий. В подобных случаях рекомендуется сосредоточить весь код, воздействующий на данную переменную, в одном действии.

Нарушение условий (2) и (3) обычно подразумевает, что программист пытается многократно перезаписывать одну и ту же переменную, предполагая, что действия шага будут выполняться в определенном порядке. Как упоминалось выше, порядок выполнения действий, связанных с одним и тем же шагом, в большинстве случаев непредсказуем, поэтому последствия данной операции также заранее неизвестны. В подобных случаях рекомендуется задать четкий порядок выполнения операций.

На рис. 15 показано два варианта решения проблемы зависимости действий, изображенной на рис. 14. Первый вариант (рис. 15a) предполагает, что все операции, связанные с нарушением условий (1), (2) и (3) будут собраны в одном действии. В варианте 2 (рис. 15b) действия разнесены по различным шагам, что позволяет установить нужный порядок их выполнения. Шаги  $Sx$  и  $Sy$  будут последовательно циклически выполняться до тех пор, пока условие  $Tx$  не станет истинным.

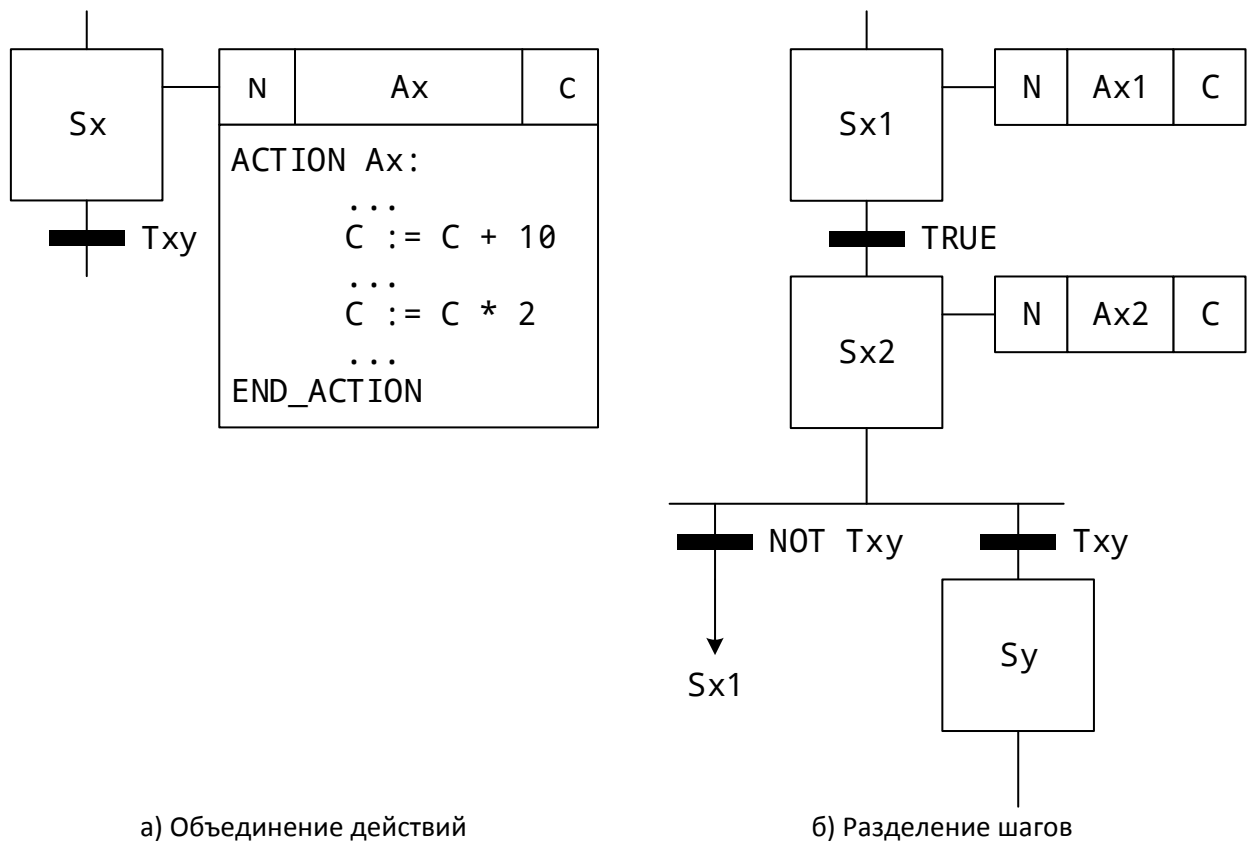


Рис. 15. Разрешение проблемы зависимости действий

Следует отметить, что представленные решения не являются эквивалентными. В варианте *a* все операции выполняются в пределах одного цикла ПЛК, в варианте *b* их выполнение занимает как минимум два цикла.

#### 4.8. Классификаторы S и R

Классификаторы S и R позволяют сделать SFC-схему более компактной – вместо привязки действия к каждому из шагов, на котором оно должно выполняться, достаточно привязать его к начальному и конечному шагу. С другой стороны, это уменьшает читабельность схемы: при большом разрыве или наличии расхожимости между шагами определить логику выполнения действия становится крайне сложно. Кроме того, тестирование, отладка и доработка программы в данном случае может вызвать затруднения. Поэтому рекомендуется ограничить использование классификаторов S и R теми фрагментами схем, в которых разрыв между начальным и конечным шагов выполнения действия является достаточно небольшим и не имеет расхожимости. Также следует помнить, что действие, вызванное однажды с классификатором S, обязательно должно быть впоследствии вызвано с классификатором R.

#### **4.9. Флаги и неявные переменные шагов**

Избегайте использования флагов и неявных переменных шагов (<имя\_шага>.X и <имя\_шага>.T) в коде действий. Это затрудняет понимание логики выполнения действий и, кроме того, может создавать проблемы в процессе разработки – например, при изменении имени шага потребуется заменить его в коде всех действий, которые используют его флаги.

#### **4.10. Независимость действий**

Избегайте зависимостей между действиями и не делайте предположений о порядке их выполнения. См. более подробную информацию в [п. 4.7](#).

## 5. Диаграммы состояний

SFC-схемы управления процессами в значительной степени схожи с [диаграммами состояний](#). Диаграммы состояний являются известным и эффективным инструментом для описания динамических систем. Они имеют несколько отличий от SFC-схем:

- В диаграммах состояний в каждый момент времени только один шаг является активным (в SFC-схеме таких шагов может быть несколько);
- В диаграммах состояний отсутствует понятие действия.

Диаграммы состояний представляют собой графическое представление [конечных автоматов](#). Удобством использования конечных автоматов является простота их описания. Для использования автомата необходима следующая информация:

- *Конечное множество состояний S*. Каждое состояние является уникальным и значимым в рамках рассматриваемой системы. В SFC состояние представляется в виде шага;
- *Множество событий E*. События обуславливают переходы системы из одного состояния в другое. В рамках SFC события можно ассоциировать с входными сигналами системы (сигналы полевого уровня, команды оператора и т.д.), а также внутренними переменными, зависящими от заложенных алгоритмов (например, результат работы таймера);
- *Начальное состояние I*. Одно из состояний множества S должно являться начальным. В рамках SFC начальное состояние (начальный шаг) обычно соответствует инициализации программы управления;
- *Множество финальных состояний F*. Эти состояния принадлежат множеству S и соответствуют завершению технологического процесса. Если автомат после прекращения работы находится в состоянии, не принадлежащем F, то это свидетельствует о некорректной работе системы управления;
- *Множество условий перехода T*. Условие перехода является функцией, связывающей три элемента:  $\langle s_i, e_{ij}, s_j \rangle$ , где  $s_i$  – текущее состояние автомата,  $e_{ij}$  – событие, определяющее переход от состояния  $s_i$  к состоянию  $s_j$ ,  $s_j$  – новое состояние автомата.

Модель выполнения конечного автомата подразумевает, что в каждый момент времени автомат находится в одном из состояний; при этом его активное состояние периодически меняются. В начальный момент времени автомат находится в состоянии I. Он остается в этом состоянии до возникновения события, которое активирует переход к следующему состоянию. Далее по мере возникновения тех или иных событий автомат будет переходить в соответствующие состояния, пока его текущее состояние не станет финальным. Таким образом, конечный автомат можно представить в виде алгоритма, входными данными которого являются события, а выходными – информация о текущем активном состоянии или некорректном завершении работы.

Модель выполнения также подразумевает, что рассматриваемая система является детерминированной – то есть в каждый момент времени находящейся только в одном состоянии. Из этого вытекает требование о том, что события, характеризующие условия перехода, должны быть взаимоисключающими.

Например, условия  $\langle si, ex, sj \rangle$  и  $\langle si, ek, sk \rangle$  не могут одновременно принадлежать множеству  $T$  – иначе такой автомат будет недетерминированным. Недетерминированный автомат не является ошибочным, но требует более сложной и менее эффективной модели поведения.

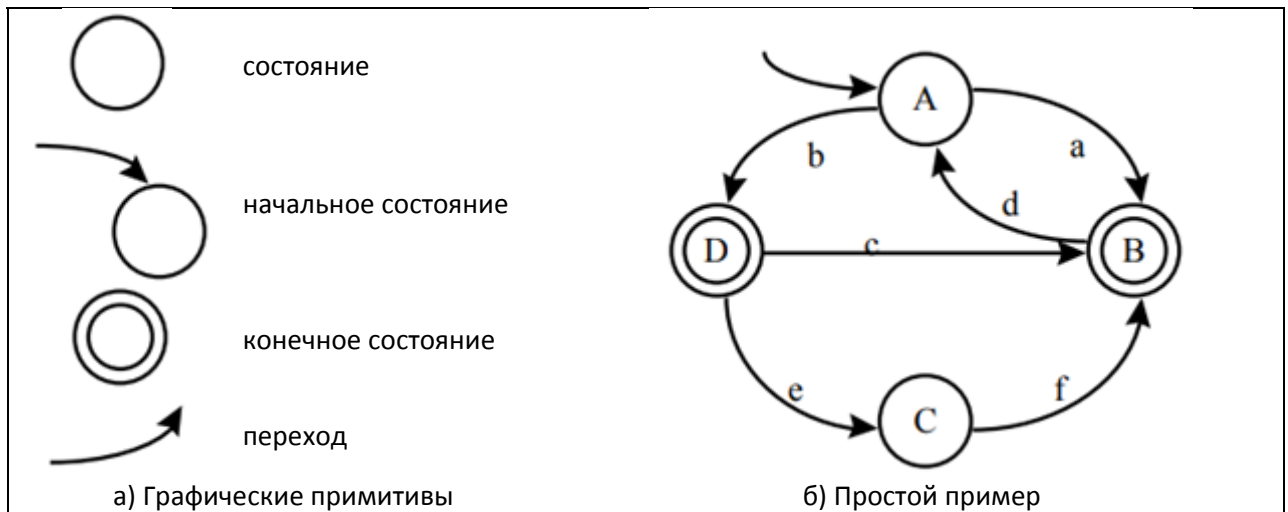


Рис. 16. Пример использования диаграммы состояний

Диаграмма состояний позволяет представить конечный автомат в виде [ориентированного графа](#). Элементы диаграммы обозначаются с помощью графических примитивов:

- Состояния обозначаются окружностями;
- Начальное состояние обозначается окружностью со стрелкой, которая не связана с другими элементами;
- Конечные состояния обозначаются окружностями с двойным контуром;
- Переходы обозначаются стрелками.

На рис. 16b приведен простейший пример диаграммы с 4 состояниями (2 из которых являются финальными) и 6 условиями перехода. Формальное описание этой системы выглядит следующим образом:

$$S = \{A, B, C, D\}$$

$$E = \{a, b, c, d, e, f, g\}$$

$$I = A$$

$$F = \{B, D\}$$

$$T = \{\langle A, a, B \rangle, \langle A, b, D \rangle, \langle D, c, B \rangle, \langle B, d, A \rangle, \langle D, e, C \rangle, \langle C, f, B \rangle\}$$

Наиболее важной задачей при создании конечного автомата является выбор состояний, характеризующих ключевые свойства моделируемой системы. В значительном числе случаев их сложно выделить из множества всех возможных состояний. Если предположить, что система описывается  $N$  переменными, каждое из которых может принимать  $M$  значений, то число состояний системы составляет  $M^N$ . Даже если некоторые состояния являются недостижимыми или незначимыми, их общее количество все равно остается крайне большим.



В связи с этим требуется выделить наиболее важные, значимые состояния системы. Представим себе погрузчик, перемещающийся по складу. Во время движения погрузчика переменные, характеризующие его положение в пространстве, непрерывно меняются. Но с точки зрения управления процессом движения значение имеют не конкретные текущие координаты погрузчика, а их соответствие заданному пути. Таким образом, все состояния, которые принимает погрузчик в процессе своего движения, можно выделить в общее состояние (Moving). С этим состоянием будет связано действие, описывающее управление движением (MotionControl).

Эффективным способом определения значимых состояний является изучение событий, связанных с взаимодействием отдельных объектов системы. В любой ситуации можно представить как минимум два таких объекта: управляемый объект и объект, с помощью которого осуществляется управление. В рамках рассмотренной выше системы управляемым объектом является погрузчик, а управляющим объектом может, например, являться оператор, производящий настройку погрузчика с панели управления.

Для представления потока событий системы в графическом виде удобно использовать [диаграммы последовательности](#). Их построение происходит по следующим правилам:

- Объекты системы изображаются вертикальными линиями;
- Взаимодействие между объектами обозначается горизонтальными линиями. Стрелки на концах линий указывают на объекты, которым передаются команды или сообщения;
- Названия событий, инициирующих взаимодействие, располагаются над линиями;
- Порядок возникновения событий на диаграмме – сверху вниз. В рамках диаграммы расстояния по вертикали между событиями не соответствуют интервалам времени между ними.

На рис. 17 приведена диаграмма последовательности, иллюстрирующая систему управления погрузчиком. Процесс управления начинается с формирования задания, которое оператор вводит на панели управления. После проверки задания на корректность оно отправляется погрузчику; оператор получает подтверждение о начале работы. После достижения точки назначения погрузчик отправляет на панель информацию о завершении перемещения. Получив эту информацию, оператор производит нужные операции (погрузка/разгрузка) и подтверждает выполнение задания. После этого управление передается автопилоту, который возвращает погрузчик на стоянку. После возвращения на панель передается сообщение о том, что погрузчик готов к выполнению новых заданий.

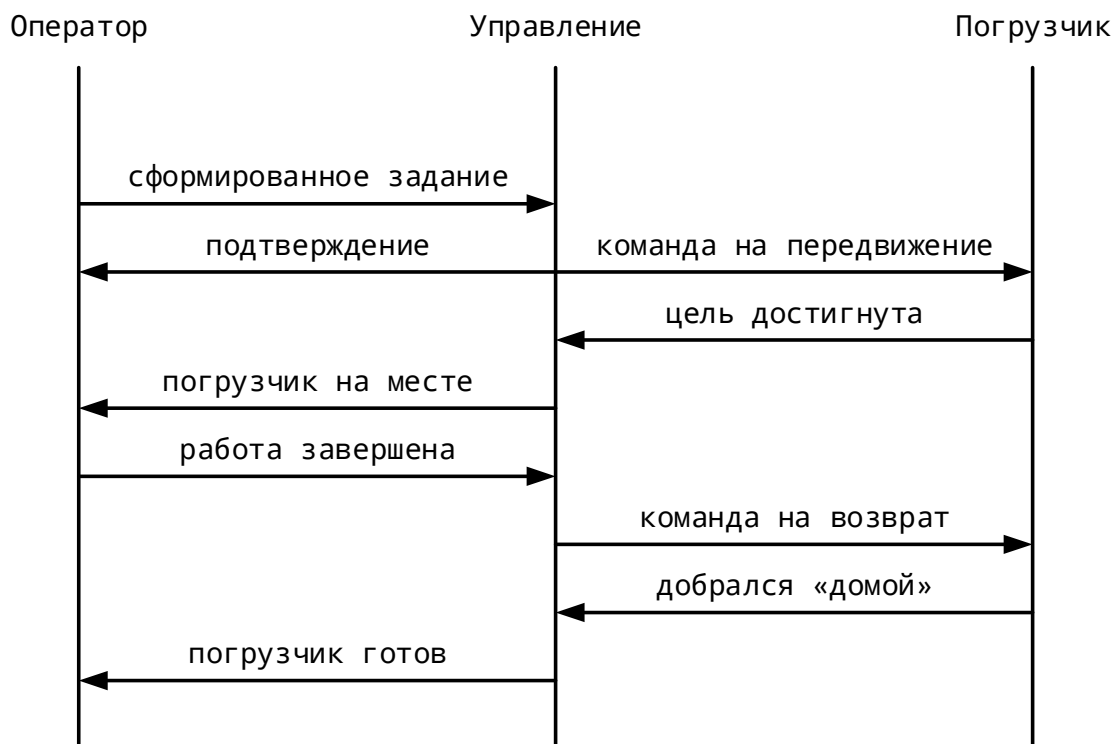


Рис. 17. Диаграмма последовательности управления погрузчиком

**Примечание:** диаграммы последовательности (с незначительными отличиями в обозначениях) также используются в [языке UML](#).

Изучив полученную диаграмму, можно легко выделить некоторые состояния погрузчика:

- *Готовность* – погрузчик находится на стоянке и готов к получению нового задания;
- *Движение* – погрузчик перемещается по территории склада;
- *Прибытие* – погрузчик находится в точке назначения;
- *Работа* – погрузчик выполняет операцию погрузки/разгрузки;
- *Возвращение* – погрузчик возвращается на стоянку после выполнения задания.

Другие состояния (не рассмотренные в примере) могут быть связаны с различного рода ошибками – например, неверно заданной точкой назначения, поломкой погрузчика и т.д.

Теперь можно перейти от формального описания системы управления к разработке соответствующей ей SFC-схемы. Каждое состояние системы будет представлено шагом. Каждый шаг будет содержать набор действий, связанных с этим состоянием (проверка корректности задания, управление движением, передача сообщений и т.д.).

Диаграммы состояний очень полезны на стадии проектирования ПО. Структура таких диаграмм крайне близка к структуре SFC-схем – но важно уметь различать эти понятия. Диаграмма является описанием системы, которая меняет свои состояния из-за внутренних или внешних факторов. SFC-схема представляет собой алгоритм, согласно которому ПЛК производит управление этой системой. Обработка SFC-схемы выполняется в каждом цикле ПЛК, что позволяет минимизировать задержку реакции на события. Переход к следующему шагу схемы осуществляется только после подтверждения о завершении текущего.

Сравнение диаграмм состояний и SFC-схем позволяет сформулировать правила, которых должен придерживаться программист для разработки корректных алгоритмов управления:

- Каждый шаг (кроме начального и финальных) начинается и заканчивается переходом;
- Каждый переход имеет только один предваряющий его шаг;
- Каждый переход ведет только к одному шагу;
- Шаг в SFC-схеме может быть расположен как до, так и после перехода;
- Для каждого перехода определено условие;
- Один шаг может быть достигнут с помощью разных переходов;
- Один шаг может быть покинут с помощью разных переходов;
- Переходы между различными шагами могут иметь одинаковые условия;
- Переходы, ведущие к одному и тому же шагу, должны иметь взаимоисключающие условия.

Синтаксические различия между диаграммами состояний и SFC-схемами могут являться причинами ошибок в ПО при недостаточной квалификации разработчика.

## 6. Примеры использования диаграмм состояний

### 6.1. Простой пример управления двигателем

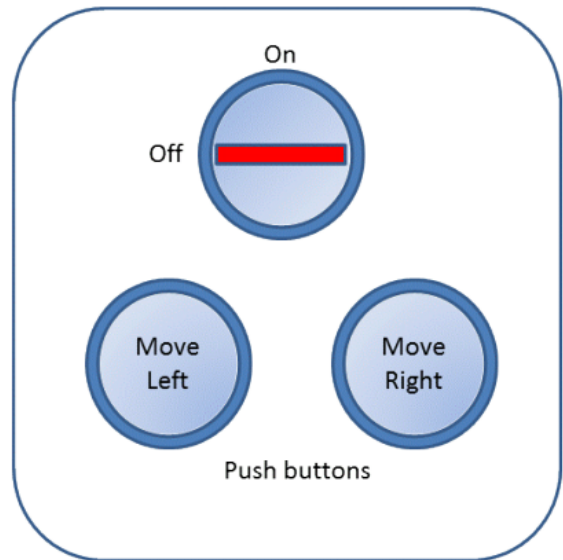
#### 6.1.1. Основная информация

Для начала рассмотрим простейший пример – управление двигателем с помощью переключателя питания и двух кнопок без фиксации. В данной системе можно выделить 3 состояния:

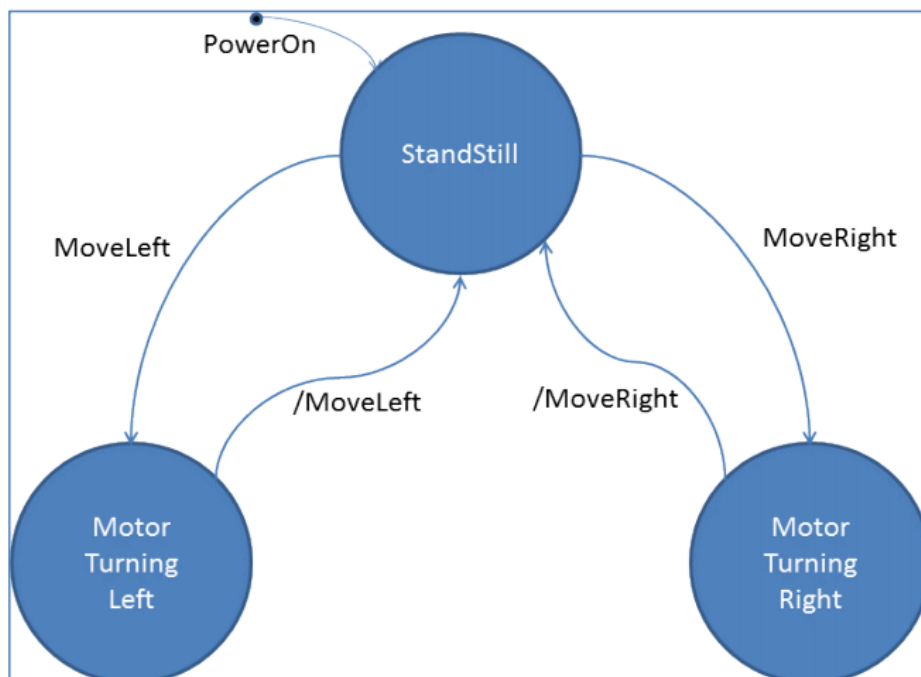
1. Двигатель включен и остановлен;
2. Двигатель вращается по часовой стрелке;
3. Двигатель вращается против часовой стрелки.

**Примечание:** в рамках примера состояние «двигатель выключен» не рассматривается.

Переход в состояние 1 (StandStill) происходит при повороте переключателя питания в положение On. Переход в состояние 2 (Motor Turning Right) происходит при нажатии кнопки Move Right. При отпускании кнопки система возвращается в состояние 1. Переход в состояние 3 (Motor Turning Left) происходит при нажатии кнопки Move Left. При отпускании кнопки система возвращается в состояние 1.

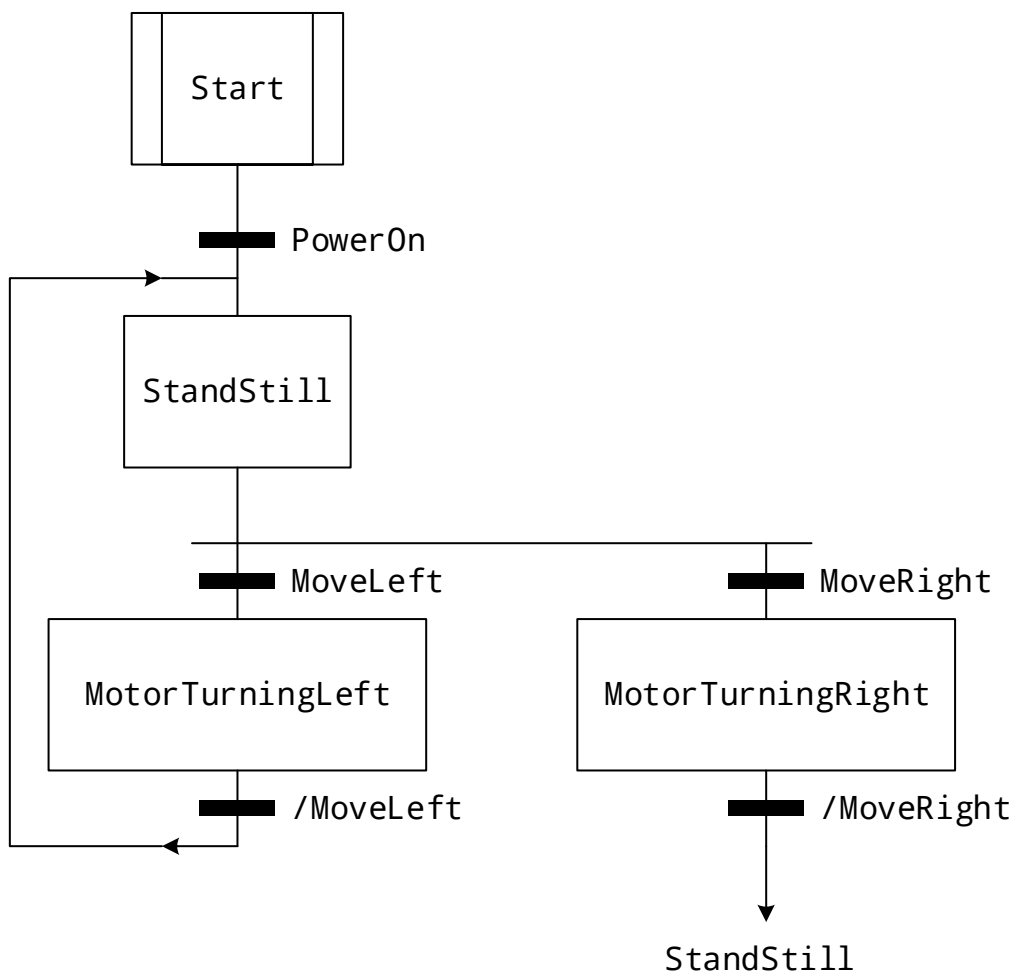


#### 6.1.2. Диаграмма состояний



### 6.1.3. Реализация на SFC

Диаграмму состояний из [п. 6.1.2](#) можно представить в виде следующей SFC-схемы:



**Примечание:** условиями переходов **/MoveLeft** и **/MoveRight** является отпускание соответствующих кнопок.

На шагах **MotorTurningLeft** и **MotorTurningRight** осуществляется управление двигателем. Например, эти шаги могут содержать действия, в которых выполняется вызов ФБ **MC\_MoveVelocity** с нужными параметрами (скорость, направление вращения). Данная схема может быть дополнена шагом обработки ошибок. Реакцией на ошибки может являться выключение и повторный запуск двигателя.

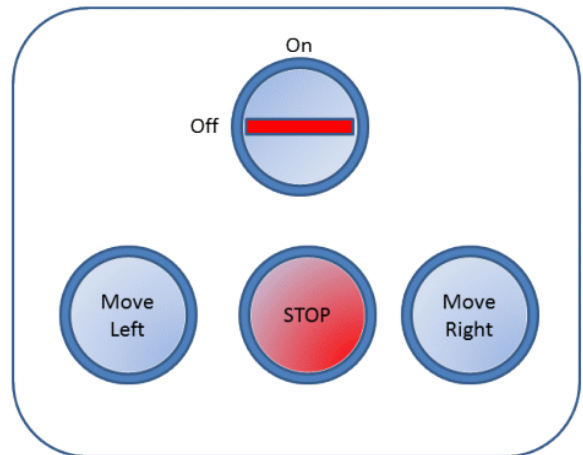
## 6.2. Расширенный пример управления двигателем

### 6.2.1. Основная информация

Дополним предыдущий пример возможностью остановки двигателя с помощью кнопки STOP. Элементы управления Move Left и Move Right в данном случае будут представлять собой кнопки с фиксацией. В данной системе можно выделить 3 состояния:

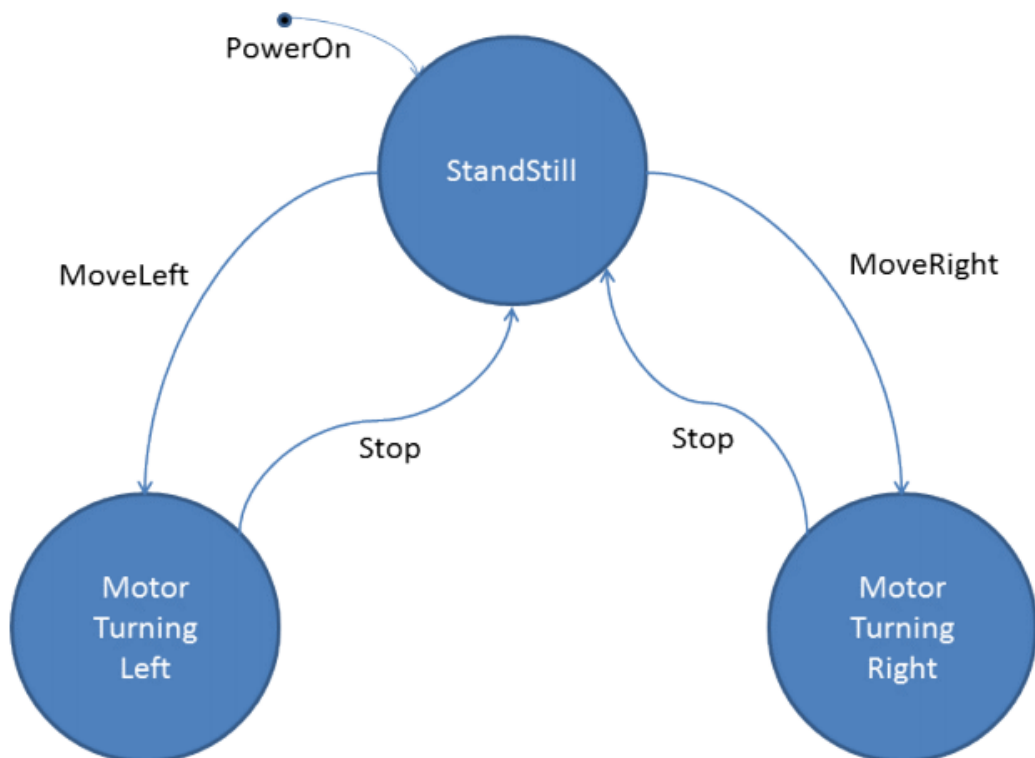
1. Двигатель включен и остановлен;
2. Двигатель вращается по часовой стрелке;
3. Двигатель вращается против часовой стрелки.

**Примечание:** в рамках примера состояние «двигатель выключен» не рассматривается.



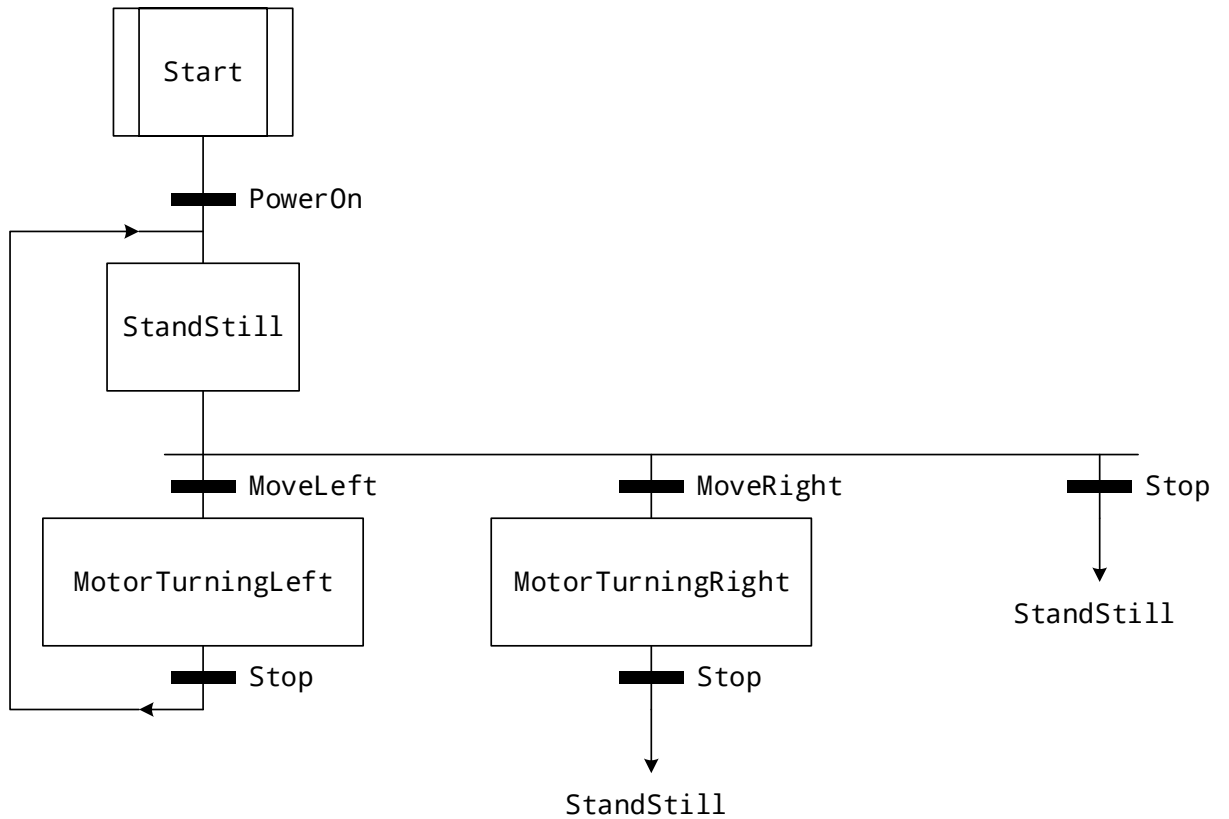
Переход в состояние 1 (StandStill) происходит при повороте переключателя питания в положение On. Переход в состояние 2 (Motor Turning Right) происходит при нажатии кнопки Move Right. Переход в состояние 3 (Motor Turning Left) происходит при нажатии кнопки Move Left. При нажатии кнопки STOP система возвращается в состояние 1.

### 6.2.2. Диаграмма состояний



## 6.2.3. Реализация на SFC

Диаграмму состояний из [п. 6.2.2](#) можно представить в виде следующей SFC-схемы:



В рамках данной схемы обработка ошибок не осуществляется. См. пример обработки ошибок в [п. 6.3.3](#).

### 6.3. Пример перевода диаграммы PackML в SFC-схему

#### 6.3.1. Основная информация о PackML

Диаграммы состояний позволяют четко определить набор функций конкретного устройства, упростить процесс проектирования и разработки ПО, а также определить общую эффективность оборудования (OEE). В качестве примера диаграммы состояний рассмотрим диаграмму языка PackML, разработанного консорциумом ОМАС для упаковочного оборудования.

Язык PackML включает в себя три основных элемента:

- диаграммы состояний;
- описание данных и соглашение по наименованию (PackTags);
- описание режимов работы оборудования.

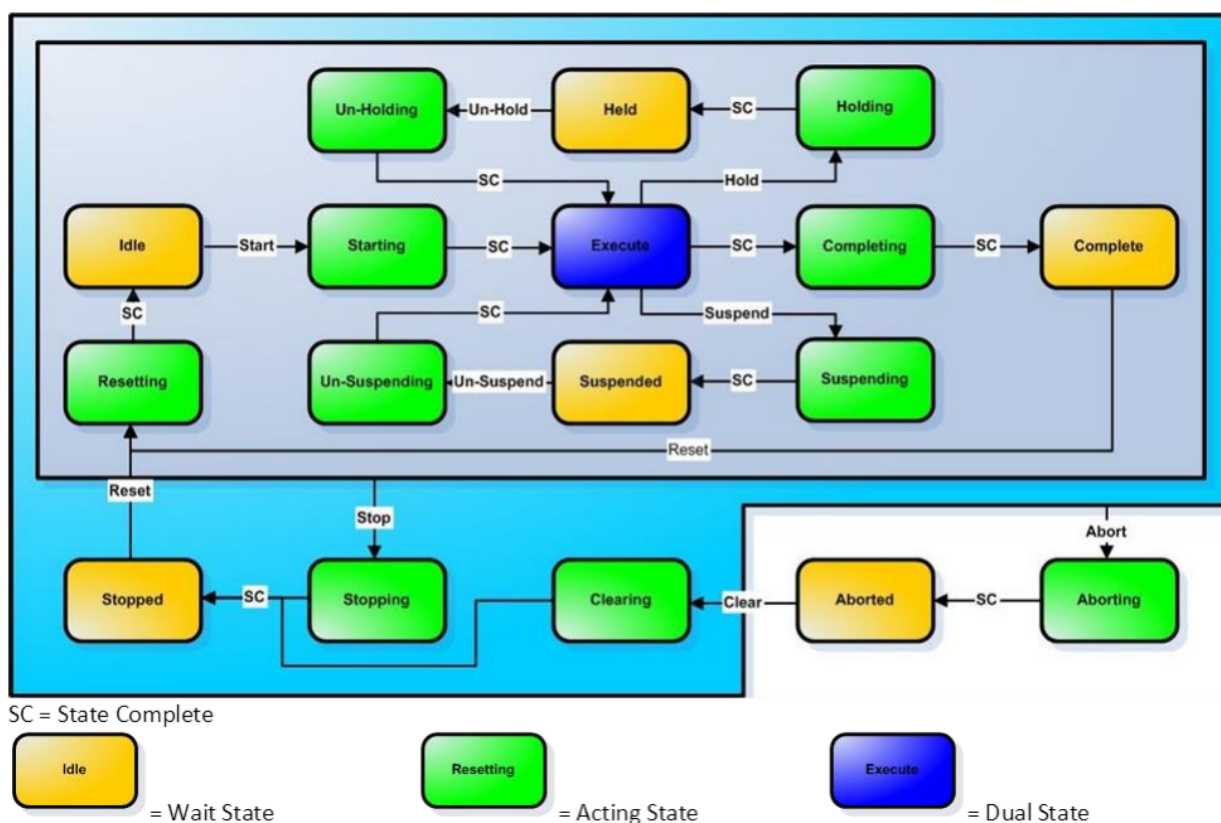


Рис. 18. Диаграмма состояний языка PackML

На диаграмме отображены три типа состояний:

- **Активные состояния** (Acting State), в которых происходит выполнение заданных операций. Прекращение операций осуществляется по истечению лимита времени или достижения определенных условий;
- **Состояния ожидания** (Wait State), переход в которые осуществляется после выполнения заданных операций. В этих состояниях не происходит выполнение каких-либо действий;
- **Двойные состояния** (Dual State), представляющие собой последовательные переключения между активными состояниями и состояниями ожидания.



Оборудование, находящееся в состоянии ожидания или двойном состоянии, может быть переключено в другое состояние только по команде оператора. Оборудование, находящееся в активном состоянии, автоматически переходит в другое состояние после достижения заданных условий (на схеме это отображается надписью SC над стрелкой перехода).

Приведенная на рис. 18 диаграмма имеет состояние **Execute**, которое можно интерпретировать как производство продукции. Цикл под этим состоянием (**Suspended**) представляет собой ожидание полуфабрикатов. Цикл над состоянием Execute (**Held**) представляет собой состояние паузы. После завершения производства очередной партии продукции система переходит в состояние **Complete** и находится в нем до поступления команды на производство следующей партии.

При включении питания система находится в состоянии **Stopped**. После команды *Reset* она переходит в состояние сброса (**Resetting**), а затем – в состояние ожидания (**Idle**). По команде *Start* система переходит в состояние **Starting**, а после него – в состояние **Execute**.

Прерывание производства происходит либо при возникновении в любом из состояний ошибок в работе оборудования (см. нижний цикл диаграммы, начинающийся с условия *Abort*), либо по команде *Stop* от оператора.

### 6.3.2. Преобразование диаграммы состояний в SFC

Диаграмма состояний представляет собой лишь описание процесса управления. Для создания алгоритма управления необходимо использовать соответствующее ПО, в котором будет осуществлена реализация этой диаграммы. Одним из вариантов реализации является использование языка SFC. Благодаря своей простоте и наглядности его не составит труда освоить инженеру любой специальности вне зависимости от уровня квалификации.

Для создания на SFC диаграммы состояний с рис. 18 потребуется реализовать следующую последовательность шагов (каждый шаг описывает одно из состояний системы): **Stopped**, **Resetting**, **Idle**, **Starting**, **Execute**. После шага **Execute** следует альтернативная расходимость с шагами **Complete**, **Hold** и **Suspend**. Шаг **Complete** является финальным. С шагов **Hold** и **Suspend** возможен переход на шаг **Execute**.

### 6.3.3. Обработка ошибок

В случае команды на остановку или возникновении ошибок, из любого шага SFC-схемы может быть осуществлен переход на шаг *Stopped*. Существует два основных варианта обработки ошибок:

- *Централизованный* – все ошибки обрабатываются на одном шаге;
- *Децентрализованный* – для каждого шага создается индивидуальный шаг обработки ошибок.

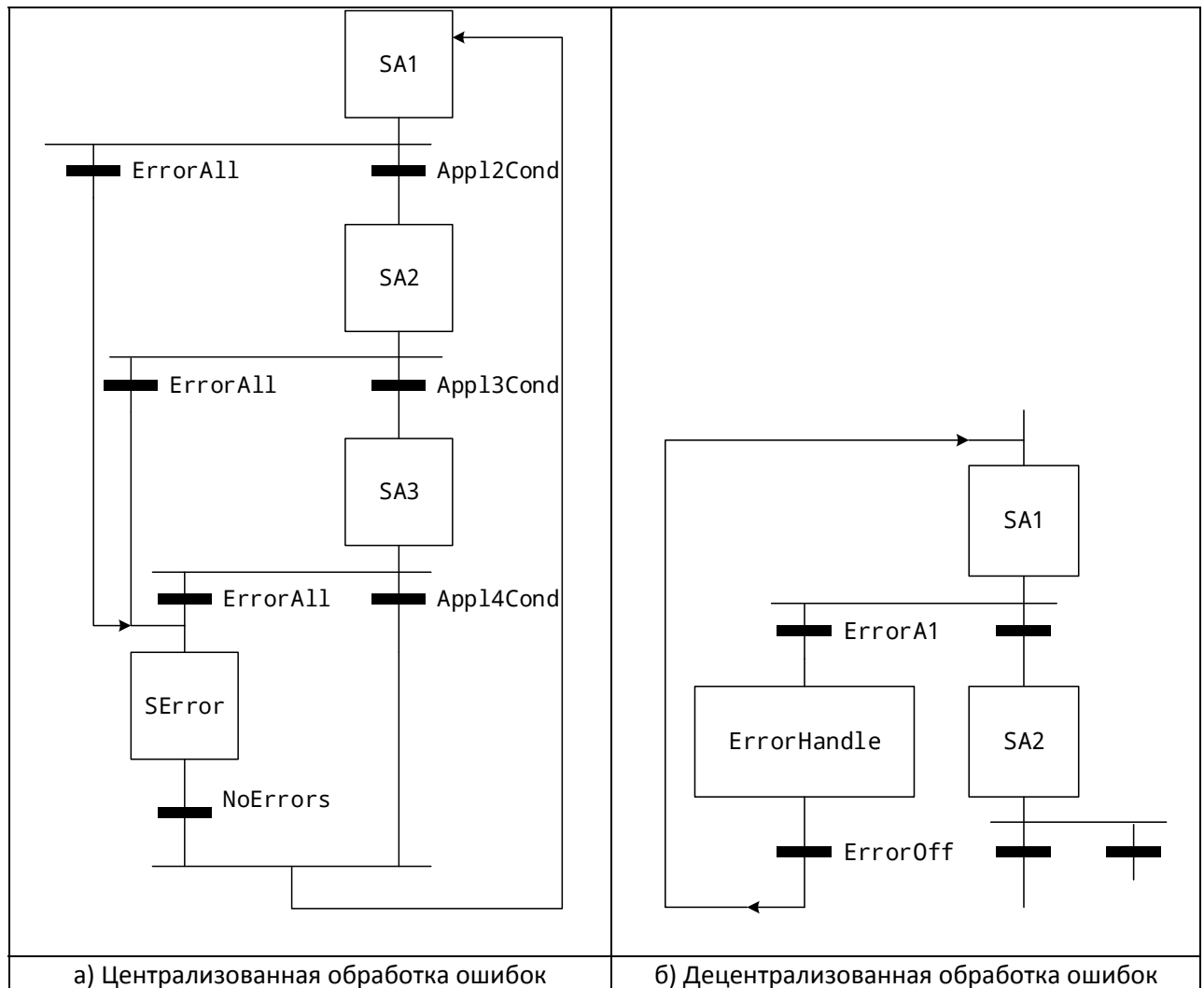


Рис. 19. Пример централизованной и децентрализованной обработки ошибок

Фрагмент SFC-схемы производственного процесса выглядит следующим образом:

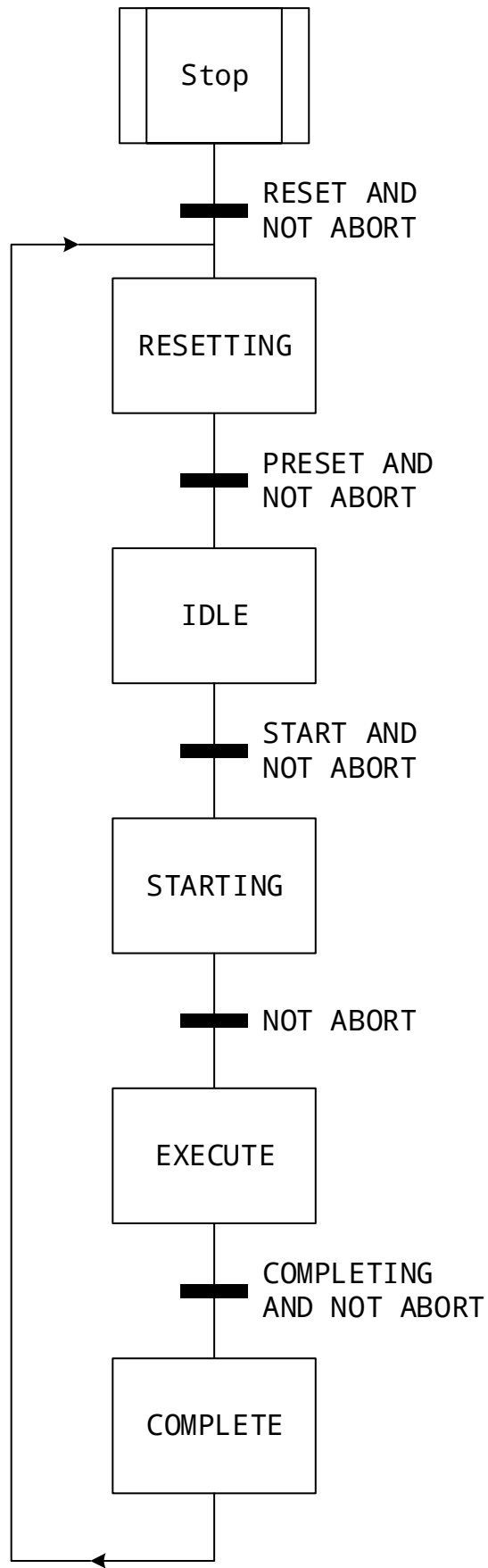


Рис. 20. Фрагмент SFC-схемы производственного процесса для диаграммы состояния с рис. 18

На рис. 21 изображена полная SFC-схема для диаграммы состояний с рис. 18, включающая обработку ошибок, возможность прерывания процесса, стадии ожидания полуфабрикатов и паузы.

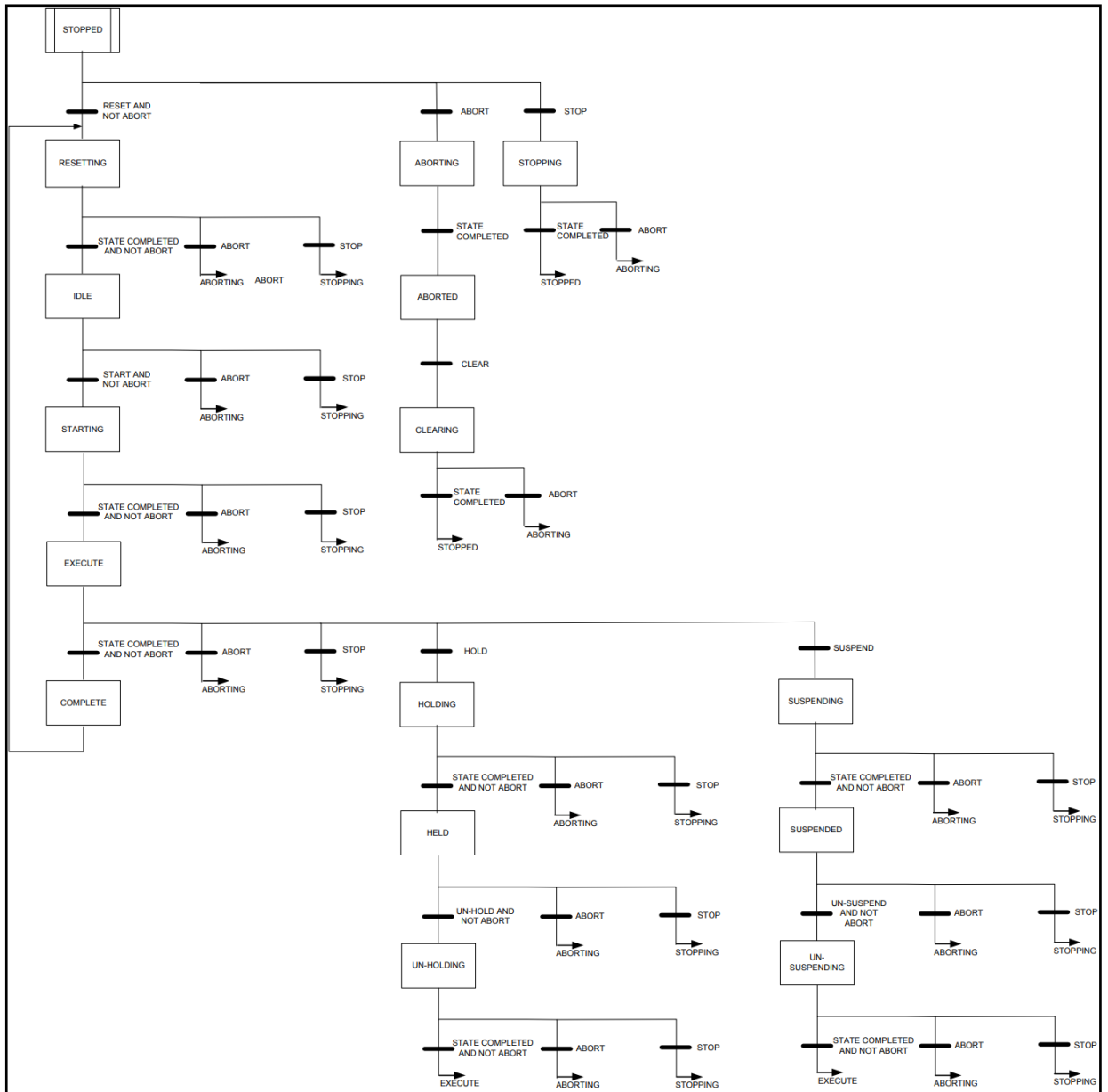


Рис. 21. Полная SFC-схема для диаграммы состояния с рис. 18

Примечание: на рисунке выше для шагов Aborting/Stopping/Resetting полные условия переходов (Start AND NOT Abort) скрыты для повышения читабельности, так как в противном случае диаграмма не вписывается в размеры страницы.

Обратите внимание, что порядок переходов может быть такой: Abort – Stopping – Stopped. Но для повышения читабельности на рисунке выше это не предусмотрено. Также важно отметить, что все условия переходов взаимоисключающие.

### 6.3.4. Аспекты безопасности для различных режимов работы

Диаграммы состояний PackML подходят для описания разных режимов работы оборудования: автоматического (этот режим рассмотрен в примере выше), полуавтоматического, режима настройки. В различных режимах система описывается разным набором доступных состояний. В автоматическом режиме доступны все состояния, и никаких специальных мер безопасности не предусмотрено. В полуавтоматическом режиме состояние паузы (*Held*) недоступно, так как полуфабрикаты подаются только по команде оператора. В режиме настройки процесс производства не может быть запущен, поэтому состояния *Execute*, *Suspended* и *Holding* недоступны. Для этого в условия переходов SFC-схемы должны быть добавлены соответствующие флаги (*SemiAutoModeSelected*, *SetupModeSelected*).

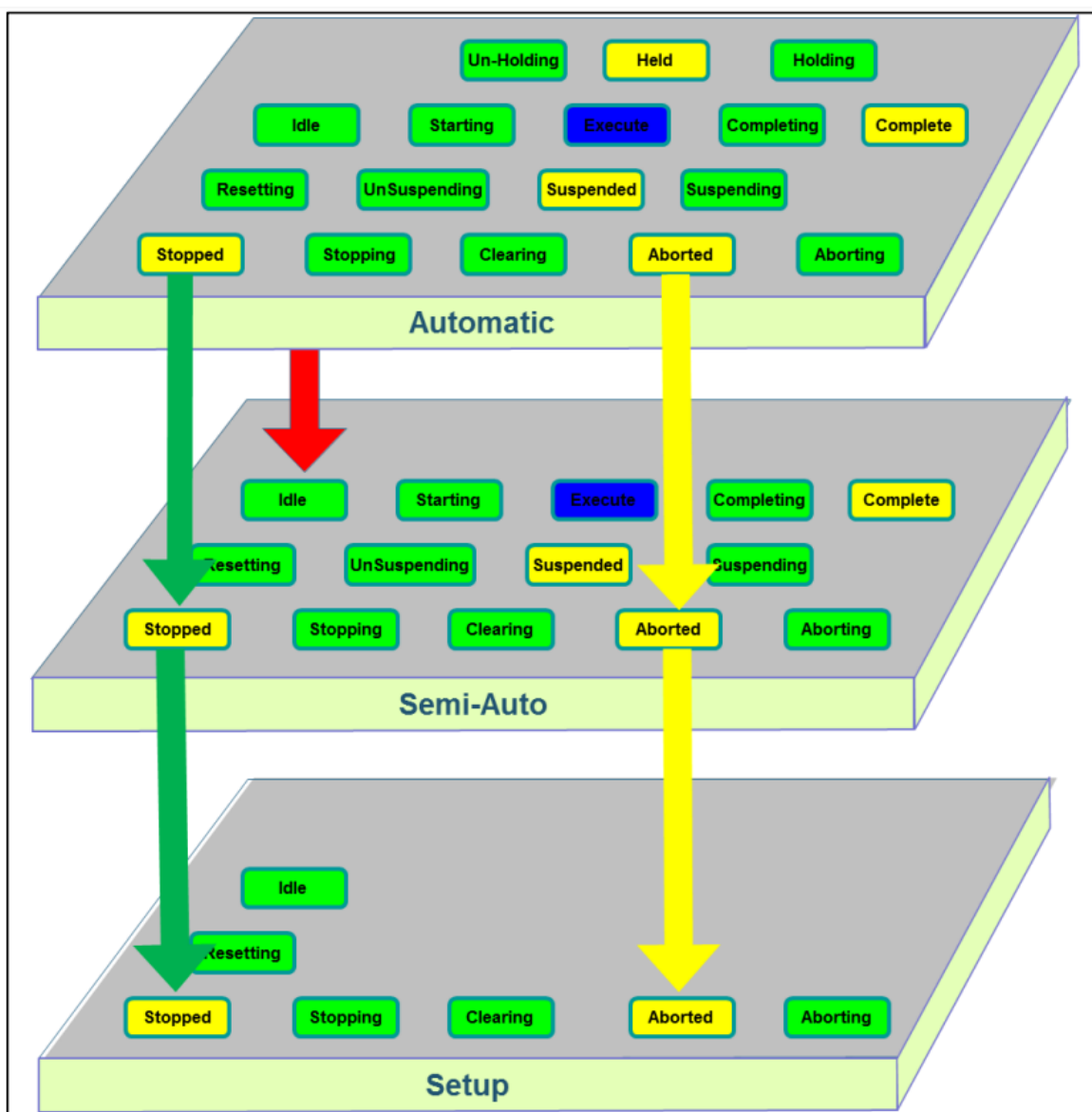


Рис. 22. Допустимые состояния для различных режимов работы

На рис. 22 приведены допустимые состояния системы для различных режимов работы оборудования.

Переключение режимов должно происходить при соблюдении требований безопасности. Например, можно реализовать в приложении селектор режимов, соответствующий спецификации безопасности PLCopen ([PLCopen Safety Specification](#)). Эта спецификация в свою очередь полностью соответствует стандарту *ANSI / PMMI B155.1-2006 Требования безопасности для упаковочных машин*.

В рамках рассматриваемого примера нет разницы в требованиях безопасности для автоматического и полуавтоматического режима. Режим настройки имеет специфические требования.

Селектор режимов позволяет детектировать потенциально опасные ситуации – например, переключение из автоматического режима в режим настройки без остановки оборудования. В этом случае селектор может произвести одно из следующих действий:

- игнорировать команду переключения режимов;
- произвести переход в состояние остановки, после чего осуществить переключение режимов;
- отдать команду на прерывание процесса и перейти в состояние *Abort*.

Реализовать контроль остановки оборудования при переходе в режим настройки можно с помощью ФБ *SF\_SafelyLimitedSpeed*, который позволяет определить допустимую скорость оборудования (например, сервоприводов) в режиме настройки и детектировать ее превышение. ФБ *SF\_SafeStop* позволяет произвести принудительный останов оборудования при выполнении заданных условий (например, возникновении ошибок). На рис. ниже приведены несколько ФБ, используемых для безопасного управления. Более подробная информация доступна в документе *PLCopen Safety Specification*, который размещен на сайте PLCopen в разделе [TC5 Safety](#).

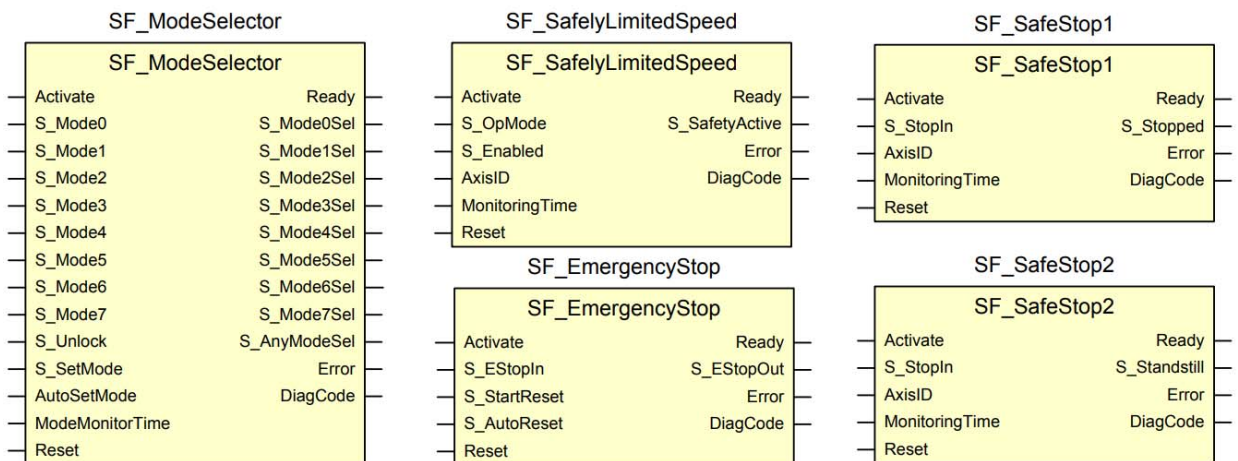


Рис. 23. Примеры ФБ из PLCopen Safety Specification

### 6.3.5. Связь SFC и сетей Петри

Возможность наличия в SFC-схеме нескольких одновременно<sup>8</sup> активных шагов позволяет провести аналогию между SFC и [сетями Петри](#). Сети Петри являются широко известным инструментом для моделирования систем с параллельно взаимодействующими компонентами. Сети Петри позволяют создать формализованное описание системы, которое упростит ее анализ.

В системах автоматизации оборудование в каждый момент времени находится только в одном состоянии. В сетях Петри несколько состояний могут быть активными одновременно, а переходы между ними осуществляются локально, затрагивая лишь часть элементов системы. Это позволяет применять сети Петри для моделирования асинхронных систем, где порядок изменения состояний заранее неизвестен. Но в контексте моделирования систем автоматизации сети Петри обладают рядом недостатков по сравнению SFC:

- акцент на недетерминированности моделируемой системы;
- отсутствие начального состояния (состояния инициализации);
- отсутствие аппарата для описания действий, выполняемых в тех или иных состояниях.

Поэтому не следует расценивать SFC как вариацию сетей Петри. Подобные сравнения могут ввести в заблуждение. Следует рассматривать SFC как средство создания детерминированных диаграмм состояний для процессов автоматизации, упрощающих проектирование прикладного программного обеспечения ПЛК.

### 6.3.6. Связь SFC и автоматов Мура и Мили

SFC-схемы управления процессами в значительной степени схожи с [диаграммами состояний](#). Диаграммы состояний представляют собой графическое представление [конечных автоматов](#). Существует два основных типа конечных автоматов:

- [Автоматы Мура](#) – в этих автоматах поведение системы определяется исключительно текущим состоянием и не зависит от предыдущего. Все рассмотренные в документе схемы относятся к этому типу;
- [Автоматы Мили](#) – в этих автоматах поведение системы может зависеть от предыдущих состояний. Автомат Мили можно преобразовать в практически эквивалентный ему автомат Мура.

Таким образом, SFC изначально рассчитан для представления автоматов Мура, но может быть использован и для представления автоматов Мили (см. [п. 4.5](#)).

---

<sup>8</sup> Т.е. активных в пределах одного цикла ПЛК (прим. пер.)